2008

Energy Efficient Image Video Sensor Networks

Paul Anthony Bender
Wright State University

Energy Efficient Image Video Sensor Networks

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

By

PAUL A. BENDER
M.S. Mathematics, Ohio University, 2004
B.S. Computer Science, Southwest Missouri State University, 1998

2008
Wright State University

WRIGHT STATE UNIVERSITY
SCHOOL OF GRADUATE STUDIES

July 14, 2008

I HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER MY SUPERVISION BY Paul A. Bender ENTITLED ENERGY EFFICIENT IMAGE VIDEO SENSOR NETWORKS BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Doctor of Philosophy.

Yong Pei, Ph.D.
Dissertation Director

Thomas Sudkamp, Ph.D.
Department Chair

Joseph F. Thomas, Jr. , Ph.D.
Dean, School of Graduate Studies

Committee on
Final Examination

Yong Pei , Ph.D.

Bin Wang , Ph.D.

Soon M. Chung , Ph.D.

Zhiqiang Wu , Ph.D.

Shih-Ta Hsiang , Ph.D.

ABSTRACT

Bender, Paul . PhD Dissertation, Department of Computer Science & Engineering, Wright State University, 2008. *Energy Efficient Image Video Sensor Networks.*

Image Video Sensor Networks are emerging applications for sensor network technologies. The relatively large size of the data collected by image video sensors presents new challenges for the sensor network in terms of energy consumption and channel capacity. We address each of these issues through the use of a high density network deployment utilizing some nodes as dedicated relay nodes.

A high density network allows network nodes to reduce their transmission power. This reduction in transmission power allows each node to conserve power and simultaneously increases the potential for spatially concurrent transmissions within the network, resulting in improved network throughputs. The use of additional relay nodes may further increase the potential for such spatially concurrent transmissions, without increasing the relay burden for each node by maintaining the same number of data generating sources in the network.

In this work, we show analytically how a high density network effects energy consumption and network capacity. We discuss the constraints placed on a high density sensor network deployment due to application latency requirements, sensor coverage requirements, connectivity requirements, and node costs.

Furthermore, we implement an Image/Video Sensor Web, an Internet enabled testbed for studying the implementation of a high density network deployment for Image/Video Sensor Networks. We utilize this testbed to verify our analytical energy results, and to study the reliable data delivery requirements necessary to successfully deploy an Image/Video Sensor Network.

Contents

List of Figures

List of Tables

Acknowledgment

I would like to acknowledge all those who have helped me through the process of completing this dissertation.

Special thanks goes to my Ph.D. Adviser, Yong Pei, and all members of my Ph.D. Committee, Bin Wang, Soon M. Chung, Zhiqiang Wu, and Shih-Ta Hsiang, for their guidance through the dissertation process.

I would like to thank the following Professors for their mentorship through the years: Yong Pei, Thomas Sudkamp, and Mateen Rizki, at Wright State University; Xiaoping Shen, Martin Mohlenkamp, Sergio Lopez-Permouth, and Carl Bruggeman at Ohio University; Eric Shade, D. Pete Sanderson, and the late Joseph Trigg Jr. at Southwest Missouri State University.

Finally, I would like to thank all my family and friends. Without your support, the many years of work required to make this dissertation a reality would have been a tremendous struggle.

Dedicated to
My parents, Richard and Kay, without whom none of this would be possible.
My fiance, Melissa, who brightens every day of my life.

Introduction

Traditional field scientists have a long standing habit of working in wilderness areas in order to observe the phenomenon or creatures they are attempting to understand. Much of this work is dangerous for the scientist, or can easily become dangerous. Additionally, when dealing with wild animals in particular, the presence of the scientists can affect the behavior of the creatures being observed.

Sensor networks provide the promise of allowing the scientist to make observations from a safe distance. In many cases, this may be possible without ever stepping foot in the habitat under observation [6]. This is particularly important for making observations on distant worlds.

Traditionally, sensor networks have been designed to transmit relatively small sensor readings back to a data sink at periodic intervals. The data returned by these networks is typically environmental data (temperature,humidity,etc), biological data (heart rate,temperatures,etc) and positional information if the node is mobile. The data returned from this type of sensor is characteristically returned at a very low data rate.

While these sensors can be used to gather quite a bit of information about various environments or the creatures that live in them, the readings don't necessarily give the scientist a complete picture of what is happening. The readings may not be usable to distinguish between two similar

events, e.g. the eruption of a volcano, or the eruption of a geyser.

In order to distinguish between two similar events, scientists may have to rely on visual and/or auditory clues to determine exactly what kind of event they are observing. To provide similar information using a sensor network, multimedia data needs to be returned to the user making observations.

Most sensor networks use a wireless communication path. This aids deployment of the sensors because no existing infrastructure is required in the area being monitored. Throughout this work, we will use the terms ”sensor network” and ”wireless sensor network” interchangeably.

1.1 Problem Statement

Our work studies a particular class of wireless sensor networks, Image/Video Wireless Sensor Networks (IVWSN), which are intended to provide visual multimedia data to an observer. Major challenges which we believe are critical to the success of this class of sensor networks include the following:

First, we must be able to address the power consumption issue. While this is a challenge for all sensor networks which utilize batteries for power, in an IVWSN power consumption must not only take into account the energy used by the communication hardware, but also the energy consumed by the sensor to capture the image. In many cases, capturing and/or compression of the Image/Video on the sensor, as required due to the limited bandwidth, will consume at least as much power as the communication hardware [4, 7].

Second, we must be able to provide sufficient bandwidth to the active sensor nodes to transmit

the data back to the data sink. While this is a requirement in other sensor networks as well, IVWSN demands more bandwidth per active sensor, which makes managing the available bandwidth more stressful.

Third, end-to-end delay has to meet the latency requirements of the application. In contrast to delay tolerant sensor applications, where storage based solutions may ease the bandwidth and energy problems, in a delay critical scenario, we need to be careful not to significantly increase the latency associated with sending each transmission through the network. This is critical for certain applications, e.g., Command and Control, where real time data is required to make appropriate decisions.

Fourth, we need to be able to provide sufficient sensor coverage. This is a particular concern because the sensors in IVWSN are generally directional in nature. Sensor coverage requirements are one of the factors in determining the minimum number of sensors which must be deployed.

Finally, we need to be concerned with maintaining communication connectivity. This is similar to the fourth concern, but from a communication point of view. The communication hardware can be either omni-directional or directional in nature. Communications connectivity is another determining factor on the minimum number of sensors which must be deployed.

In this paper, we will address primarily the first 3 challenges.

1.2 Related Works

A significant body of work exists examining various ways to increase the lifetime of a Wireless Sensor Network. Much of this existing work assumes the communication hardware consumes

most of the power available to a node during its lifetime. These techniques either assume that the energy consumed by the sensors is negligible, in comparison to communication, or provide no consideration to maintaining a specified bandwidth level. Many of these techniques allow the sensor nodes to turn off their communication hardware, which effectively puts the node to sleep[8, 9, 10, 11, 12, 13].

Having the sensors sleep implies that either the data collected by nodes has to be held until a neighbor node wakes up, or the node has to increase its transmission power for the duration of the period for which it is active. The first behavior increases the latency of the received data. The second behavior artificially reduces the available node density which may unnecessarily increases the energy consumption for the active nodes, decreases the sensor resolution, and decreases the potential for parallel transmissions. The decrease in potential parallel transmissions results in a reduction in network throughput and may also cause a reduction in network lifetime due to increasing packet collision.

Other power saving methods presented in existing literature include workload sharing methods and network hardware based topology control. The workload sharing methods, e.g., [14] and [15], attempt to share the sensing, computing, and communication workload among all the sensors. Hardware based topology control adapts features of the network hardware, such as the power consumed by the transceiver [16] and the use of tunable or directional antennas.

We further explore methods described in existing literature in Section 2.1.

1.3 Our Approach

As pointed out by Xing et. al in [17], when traffic levels are high, having nodes sleep may increase the energy required for transmission. Additionally, there is a significant amount of energy consumed to coordinate sleeping.

Since we expect traffic levels to be high in an IVWSN, we believe new approaches are required to accomplish our goal. We propose two approaches by which the lifespan of an IVWSN may be increased without adverse affects on the ability of the network to deliver multimedia data in a timely manner.

Firstly, we believe it is crucial to reduce the transmission energy consumption by reducing the transmission distance in order to conserve energy for image capture. We propose to increase the density of sensors in the IVWSN beyond what is required to provide connectivity and sensor coverage. This dense deployment allows reducing the transmission power, which provides a direct decrease in power consumption as shown in Section 3.2. Additionally, reduced transmission power indirectly reduces the power consumption by reducing the number of messages the adjacent sensor receives but discards because it is not on the communication path selected by the routing protocol in use. This phenomenon is discussed further by Dong in [18].

The reduction in power level allowed by the density increase has an additional benefit of increasing the the average throughput available to each node. This is due largely to increased spatial reuse of the available communication channel[19].

Secondly, we believe it benefits to distinguish between sensing density and communication density. We propose the use of some nodes strictly for relaying data. These relay nodes are either

normal sensor nodes on which the sensor hardware has been turned off to conserve energy when high resolution surveillance is not necessary, or nodes which are deployed without the sensor hardware to reduce cost.

When the nodes acting as relay nodes are fully equipped sensor nodes, we will rotate which nodes are actively taking readings. When a node is not actively taking readings it acts as a relay for neighboring nodes. In order to provide consistent sensor coverage, we need to coordinate which nodes are allowed to turn off their sensor hardware. Since our nodes never turn off their communication hardware, the open communication channel may be utilized for this coordination. We believe this leads to an advantage over other energy saving schemes [14, 11, 12, 13] because it eliminates control overhead associated with turning communication equipment on and off, and it alleviates any need for predeployment configuration [8, 9, 15, 10], allowing adaptive sensing. When dedicated relay nodes are deployed, we use the additional nodes strictly to relay data for the sensor nodes.

When relay nodes are used, our analytical results show the capacity of the network is actually increased despite the additional nodes. This occurs because nodes acting as relay nodes do not generate any traffic of their own.

Moreover, the proposed approaches may also better satisfy the delay requirements of data delivery in multimedia sensor networks when properly designed.

Full analysis of our dense network deployment is discussed in Chapter 3.

1.4 Implementation

We have implemented an Image/Video Sensor Network Testbed which can be utilized to study the effects of a high density deployment on the 5 constraints presented in Section 1.1. This testbed consists of several sensor nodes capable of transmitting images through a network of specified density to a sink node.

The sink node utilized for our experiments includes a web server for dissemination of the received images to monitoring systems via the Internet. A collection of identical networks can be deployed in multiple locations. Each network may be monitored from one or more location simultaneously. This collection of sensor networks and monitoring systems forms a Sensor Web.

A complete description of our implementation appears in Chapter 4.

Related Work - Literature Reviews

2.1 Existing Power Saving Methods

Existing literature is filled with many examples of how to conserve energy in a power limited wireless network. These methods can be divided into two general categories. First, there are methods which divide the workload among the nodes. Second, there are methods which control the topology of the network. The topology control methods can be further divided into algorithms that control the topology by allowing nodes to sleep, and algorithms which adjust the network structure by manipulating features of the underlying network hardware. Features exploited in this second topology control method include controlling the power consumed by the transceiver and the use of tunable or directional antennas.

2.1.1 Workload Sharing

Workload sharing methods assume that the network's load can be balanced among the nodes over time.

2.1.1.1 Optimizing Energy Consumption in Wireless Ad Hoc Networks

One workload sharing method example is presented by A. Abbas et al. in [14]. In the method presented in this paper, tasks are traded between the nodes of the network in an attempt to balance the load associated with each node.

Simulation results provided in the paper indicate a network longevity increase of between 3% and 40% .

For our application, this method is unusable. The primary issue is that the work each of our multimedia sensor nodes undertakes involves handling data generated by sensors, which are triggered by location specific events.

2.1.1.2 Energy Efficient Strategies for Deployment of a Two-Level Wireless Sensor Network

In [15], the authors discuss methods by which energy can be saved by dividing a sensor network hierarchically.

The network is divided at deployment time into sensor nodes and micro-server nodes. The sensor nodes are connected by a few hops to a micro-server node. The micro-server nodes are connected by a few hops to the data sink.

The paper derives an analytical model for determining the theoretical maximum lifetime of 2 different network structures. The first network structure is a linear network, where the nodes are placed equidistant from their neighbors. The second network is a 2 dimensional network for which all sensor nodes are 1 hop from their assigned micro-server, and each micro-server is one hop from the data sink.

The paper concludes by providing numerical results derived using Matlab for the 1 dimensional network and simulation results for the 2 dimensional network. In each case, the numerical results indicate an increased lifetime of between 2.5 and 3 times the nominal network lifetime.

The primary issue with issue with this particular method is it requires configuring the network prior to deployment. This does not allow the network to adapt to changing conditions. The SPAN algorithm, covered in 2.1.2.1, provides a similar, though dynamic, network structure.

2.1.2 Topology Control with Sleeping Nodes

Two methods have traditionally been proposed for saving energy by allowing nodes to sleep. The first technique relies on information derived from the application layer to decide when it is acceptable for a node to sleep. The second technique is to chose a subset of the nodes allowed to sleep based on graph theoretical properties of the underlying network.

2.1.2.1 SPAN

The SPAN algorithm presented in [20] is designed to maintain a backbone through which traffic can flow. It does this by allowing a local network node to act as a coordinator for it's neighbors.

When acting as the local coordinator, the node is designated as the only node which will forward data to the nearest neighboring coordinators. This effectively segregates local traffic from non-local traffic. The coordinator functions in much the same way as a router does in wired networks. The node may withdraw from being the coordinator if every pair of it's neighbors is able to reach each other through some other neighbor, even if the new intermediate node is not currently acting as a coordinator.

When not acting as the local coordinator, the node is allowed to operate in a reduced power mode. In this reduced power mode, the node periodically wakes up to send data to the current coordinator and to check and see if it should become the local coordinator. The node will assume coordinator duties if two of it's neighbors cannot communicate directly or through one or two existing coordinators.

The authors have determined experimentally that SPAN can increase network lifespan by a factor of 2. In prior work, [21], the authors indicated a theoretical network lifespan improvement by a factor of 3.5.

For our purposes, SPAN leads to a network which increases the latency of traffic more than desirable. When a coordinator node withdraws from acting as a coordinator, there is no guarantee that each of it's neighbors can communicate with each other, since there is no requirement for an active coordinator at the time of withdrawal. This delays data from neighbors of the former coordinator until a new coordinator is selected.

SPAN, can also lead to an imbalance of workload, since there is no guarantee that the nodes will share the coordinator responsibilities equally. This results in premature resource exhaustion for some nodes. This workload imbalance may lead to a premature disconnection in the network.

2.1.2.2 Adaptive Energy-Conserving Routing for Multihop Ad Hoc Networks

In [12], two routing protocols are presented for saving energy.

The first algorithm, the Basic Energy-Conserving Algorithm (BECA), has each node working independently to conserve energy. While there is no network traffic, the nodes alternate between a sleeping and listening mode. If traffic arrives while the node is in listening mode, it becomes

active for a set period of time before returning to the sleeping state. Analytical results presented in the paper indicate BECA provides a theoretical maximum energy savings of 50% per node,but simulation results in the paper provide for a per node maximum of 35% .

The second algorithm, the Adaptive Fidelity Energy-Conserving Algorithm (AFECA), takes advantage of the spacial redundancy provided by the network density. AFECA is an extension of BECA where the sleep time is estimated based on the number of neighbors each node has. Analytical results presented for AFECA indicate a potential energy consumption reduction for each node of $2/(2 + N)$, where N is the size of the neighborhood. Simulation results indicate a maximum energy savings of 45% per node.

Simulation results presented indicate the overall network lifetime is increased by approximately 20% using BECA and 55% using AFECA. Furthermore, the paper acknowledges that AFECA takes advantage of increased density by increasing the longevity of the network as a whole.

In our multimedia case, allowing the routing nodes to sleep leads to an increase in latency. Allowing the nodes to sleep means the nodes are not available for routing purposes, so the data is either delayed until a node wakes up or a new path through the network is discovered by the routing protocol. Therefore, neither algorithm presented in [12] is suitable for our application.

2.1.2.3 Geography-informed Energy Conservation for Ad Hoc Routing

Using the geography of the network to determine which nodes are redundant, and can therefore be put into sleep mode, is explored in [13].

The algorithm presented in this paper, Geographical Adaptive Fidelity (GAF), divides the plane in which the network resides into a virtual grid. In the cells of this grid, each node is equiva-

lent for routing purposes. This means that if any one node in the cell is active, communication can occur between the two adjacent cells.

Once the grid structure is established, the nodes in each cell alternate between acting as the active relay node in the cell and sleeping. The order in which nodes are active is negotiated during a discovery phase. The discovery phase may repeat periodically, particularly when nodes are mobile.

The paper presents two versions of the protocol, a basic protocol (GAF-b) and a mobility adaptive protocol (GAF-ma). The two protocols differ in how they treat networks with high mobility. GAF-b tends to have slightly fewer active nodes than GAF-ma does. In terms of energy consumption, the two protocols behave the same when mobility is low, but when mobility is high, GAF-b outperforms GAF-ma slightly.

Simulation results provided in the paper indicate GAF will decrease per-node energy consumption by 40% to 60% . This indicates an increase in network lifetime of approximately 2.5 times the basic lifetime of the network is possible. Further simulations indicate that using GAF-b increases the base lifetime of the network by 4 times, and GAF-ma increases the base lifetime of the network by 3 times under high mobility conditions.

This paper also provides simulation results when the network density is increased. The paper indicates that when the network density is increased 4 fold, that GAF increases the base network lifetime by 4 to 6 times.

GAF's primary drawback, from our multimedia sensor network point of view, is that it reduces the density of the network. As we show in section 3.3, decreasing the density leads to a decrease in network capacity, due to reduced potential for concurrent transmissions. There may also be some latency issues during the transition phase between two nodes in the cell acting as the active relay

node for the cell in question.

2.1.2.4 Graph Theoretic Algorithms for Energy Savings

In [8], Abrams et al. describe several methods for choosing which nodes to sleep based on building a Set K-Cover using the underling network structure. The idea is to divide the plane the network occupies into several regions, and to attempt to find a maximal number of subsets of nodes for which all regions are represented. Nodes are allowed to sleep if they are not part of the currently active cover.

The regions used to divide the network are defined based primarily on sensor coverage, i.e. nodes containing sensors which cover similar areas are included in the same region.

Since the Set K-Cover problem is NP-Complete, the paper presents several approximation algorithms for the Set K-Cover problem. The claim is made that if k covers can be found which cover 80% of the regions in the network, then the network will see an increased longevity of $k-1$ times the original network lifespan.

With this method, as with the previous method, increasing the density of nodes in the network increases the longevity of the network. This is due directly to an increase in the number of nodes from each region which can be used to maintain network connectivity.

Similar energy savings methods are presented in [10] and [9]. All three methods share the common theme of using a graph theoretical property of the network to divide the network between active and sleeping nodes. [10] and [9] both create dominating sets to determine which nodes are active in the network.

The drawback to the methods presented in this section is that the graph theoretic problems used to construct the active network backbone are NP-Complete. As a result, it is computationally intensive to find a solution to the problems used. Furthermore, the best available solutions are approximations to the optimal solution, therefore it is not possible to obtain optimal energy savings in any arbitrary sensor network.

2.1.2.5 STEM

In [11], Schurgers et al. present a topology management scheme called STEM (Sparse Topology and Energy Management). STEM allows the network to be constructed in a manner by which Energy constraints and Latency constraints can be traded.

This method is similar to the methods presented in [12] in that it alternates between time spent sleeping and time spent in an active state. Where STEM differs is in how the nodes transition out of the sleeping state. There are two possible triggers for the transition. First, the nodes will wake themselves up if an on-board sensor event requires action from the CPU. Second, the nodes will wake up if another node pages them.

The node pages are handled either via a secondary communication channel, typically an ultra low power radio receiver, or through waking up the primary communication channel periodically to check for new traffic.

In addition, there are two versions of the STEM protocol. STEM-B wakes up neighboring nodes using periodic beacon messages. STEM-T wakes up neighboring nodes using a single wakeup tone.

STEM can be combined with topology management protocols, such as SPAN or GAF, to

take advantage of increased node densities. The authors claimed when this combined approach is taken the per node energy consumption can be reduced to 1% of it's nominal engery consumption, resulting in up to a 100 times increase in the network lifetime.

For our needs, STEM introduces added latency to the communication channel when nodes must wake up to handle relay traffic. Without a secondary communication channel, each hop may have to wait for the next node in the communication path to wake up. The secondary paging channel may reduce this latency, since it can potentially wake up more than one node on the communication path at a time.

This sparse traffic assumption lead the authors of [11] to conclude the dramatic energy savings presented in the paper were possible. However, from our multimedia sensor network point of view, the sparse traffic assumption made by the authors does not always apply, so STEM cannot be applied to our work directly.

In our work, a technique similar to STEM's paging of adjacent nodes may be useful for determining when a node should reactivate it's sensor.

2.1.3 Topology Control Through Networking Tuning

Tunable network parameters can be adjusted to allow some or all nodes to communicate with a larger or smaller subset of their neighbors as required. Tunable parameters may include transceiver power adjustment or adjustment to the directionality of antennas.

2.1.3.1 Topology Control of Miltie Wireless Networks using Transmit Power Adjustment

In [16] the authors discuss using adjustment of transmission power to control the network topology. The paper presents 4 algorithms for controlling the power level required to maintain connectivity in the network.

The first pair of algorithms, CONNECT and BICONN-AUGMENT, determine optimal solutions when the network nodes are static. The primary difference between CONNECT and BICONN-AUGMENT is that the CONNECT algorithm just attempts to maintain connectivity, while the BICONN-AUGMENT algorithm attempts provide a biconnected network for improved robustness.

Both of these algorithms are centralized algorithms. The authors present proofs that the CONNECT and BICONN-AUGMENT algorithms produce optimal solutions and run in $O(n^2)$ time.

The second pair of algorithms are distributed algorithms designed to maintain connectivity when the network nodes are allowed to move. The goal of each algorithm is to set the transmit power on each node such that connectivity to some threshold number of nodes is maintained. Both algorithms utilize information provided by the routing protocol to determine the number of nodes with which it can communicate, which introduces no additional load to the network.

The first of these algorithms, Local Information No Topology (LINT), uses locally available neighbor information to determine the number of nodes to which it can communicate. Periodically, the node checks to see the number of neighbors it can communicate with. If the neighbor count is too high, the node decreases power if possible. If the neighbor count is too low, the node increases power if possible.

The second in this pair of algorithms, Local Information Link-State Topology (LILT), uses link state information provided in the routing protocols to determine when it is permissible to reduce the transmission power.

LILT was designed to overcome a problem introduced by LINT. Since LINT does not use any topology information provided by the network, it can cause the elimination of a link which is required to maintain connectivity in the network.

With the LILT protocol, if a node determines the network is disconnected, it increases it's transmit power to the maximum possible. If the node determines the network is in danger of being split due to the loss of a single link, the node will increase power in order to provide biconnectivity.

These additional controls allow LILT to be resilient to mobility, provided the nodes in the network do not move beyond the maximum transmission range of the network hardware.

For our high density multimedia application, the algorithms presented in this paper provide mixed results. The CONNECT and BICONN-AUGMENT algorithms are not usable for our multimedia application due to the centralized nature of the algorithms. LINT or LILT may be beneficial to determine power settings for adjustable transceivers on our multimedia sensor nodes.

2.2 Multimedia Sensor Network Applications

Current multimedia sensor networks, include DeerNet [6], VigilNet [1], and Georgia Tech's Wireless Multimedia Sensor Network testbed [2]. These networks tend to be organized in a manner similar to the static network described in 2.1.1.2. In this case, there are a limited number of multimedia sensor nodes dispersed in a sensor network which also contains many other low level sensor

nodes. Each of the low level sensor nodes contains a low data rate sensor, e.g. a motion detector. In operation, when an event is detected by a low level sensor, the multimedia sensors will react and deliver the desired audio/image information back to the sink.

2.2.1 DeerNet

DeerNet [6][22], is an NSF sponsored project which is being conducted jointly by the University of Florida, the University of Missouri-Columbia, and the Missouri Department of Conservation. One stated use for the information generated by the project is to reduce the number of collisions between deer and automobiles.

The primary multimedia data provided by DeerNet is the video captured from the point of view of a White Tailed Deer. This information is captured by video cameras mounted to the antlers of the bucks, and to the necks of the does.

To deliver the video back to the recording station, each deer also carries a wireless transmitter and a battery pack good for several hundred hours of video transmission. In order to conserve energy, the cameras are equipped with light sensors, such that the camera is turned off in low light conditions.

The recording station consists of a 10 meter antenna and a TV/VCR combination. The video sent back from the deer may be viewed in real time and/or recorded by the VCR. The video tapes allow delayed analysis of the information. This recording station plays the same roll as the data sink in other networks.

The available information about DeerNet leads us to believe all communication occurs over

a single transmission hop using existing wireless video transmission hardware. It is unclear if the the video signal is broadcast using technology from the broadcast television industry or consumer grade wireless video transmission hardware broadcasting in unlicensed bands.

In either video transmission case, the existing technologies limit the transmission of video to a fixed number of channels. This limits the scalability of the network, since the number of simultaneous transmissions is limited to the number of available channels. For each simultaneous transmission, some or all of the equipment used in the recording station may need to be duplicated.

DeerNet is an example of the type of sensor network application which inspires our work. This thesis plans to provide improved technologies to unleash the capabilities of sensor networks for wildlife conservation and management.

2.2.2 VigilNet: An Integrated Sensor Network System for Energy-Efficient Surveillance

The authors of [1] explore the implementation of a vehicle tracking system using MICA2 motes equipped with magnetometers. The target application is for military command and control. The sensor nodes are allowed to enter a low energy consumption state periodically, but are awakened by sensor events and/or on a timed basis.

The sensor field is divided into groups. One node in a group is referred to as a sentry nodes. The sentry node is active for some fixed period of time, during which it coordinates activities of its neighbors. It also maintain a clock for intergroup synchronization purposes. When an event of interest occurs, the sentry node also acts as a relay for its neighbors. The Sentry node may change

from one node to another based on sensor coverage, remaining energy, and the presence of existing sentry nodes.

When an event of interest is detected, the sensor nodes need to communicate with each other in order to track the object. In the test case presented in the paper, the magnetometers are used to obtain positional information required to track a car traveling down a road, as depicted in Figure 2.1. This positional information is then uploaded to a data sink, a MICA2 mote attached to a laptop in this case, for further processing.

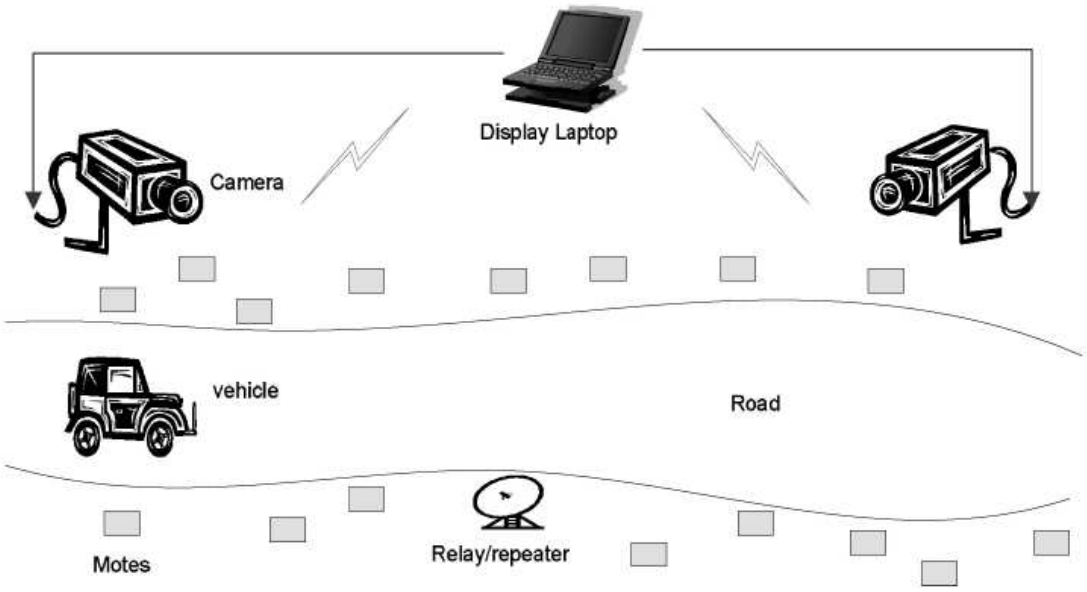


Figure 2.1: VigilNet test deployment[1]

The magnetic sensor field is also capable of triggering video camera inputs. In their setup, the video cameras are directly controlled by the laptop, but the sensor network tells the laptop when to turn the cameras on and off.

We view the surveillance application explored in the VigilNet application as another ideal application for a multimedia sensor network. We believe a sensor network is more effective, and

more scalable, when used for surveillance, if the sensor nodes control the multimedia sensors directly, rather than letting the data sink do so as implemented in VigilNet. This approach allows the sensor network to respond to events requiring attention from the multimedia sensors on a localized level, providing improved response time, due to being able to refocus the sensors without communicating with the data sink.

VigilNet does not address deployment of the multimedia sensors in sufficient density for good sensor coverage, nor does it address efficient handling of multimedia data or multimedia sensors.

2.2.3 Testbed for Wireless Multimedia Sensor Networks at Georgia Tech

Georgia Tech's Broadband and Wireless Networking Laboratory has developed a Testbed for Wireless Multimedia Sensor Networks [2]. The constructed network is designed to emulate several potential deployment models for multimedia sensor networks. The resulting network provides a hardware platform for testing multimedia sensor network algorithms. The network consists of several sub-networks, each containing a different arrangement of sensors. The architecture is shown in Figure 2.2.

The first subnetwork consists of several components arranged in layers. First, there is a field of scaler sensors. These scaler sensor nodes are MICAz motes [4] with low data rate sensors, such as motion detectors. When the scaler sensors detect an event, the sensor nodes forward information to the second layer of components, multimedia sensors nodes.

The multimedia sensor nodes consist of a camera connected to a mote or a Stargate node [23]. The multimedia sensor nodes used by Georgia Tech are computationally more powerful than the scaler sensor nodes. There are two types of multimedia sensor nodes in Georgia Tech's testbed.

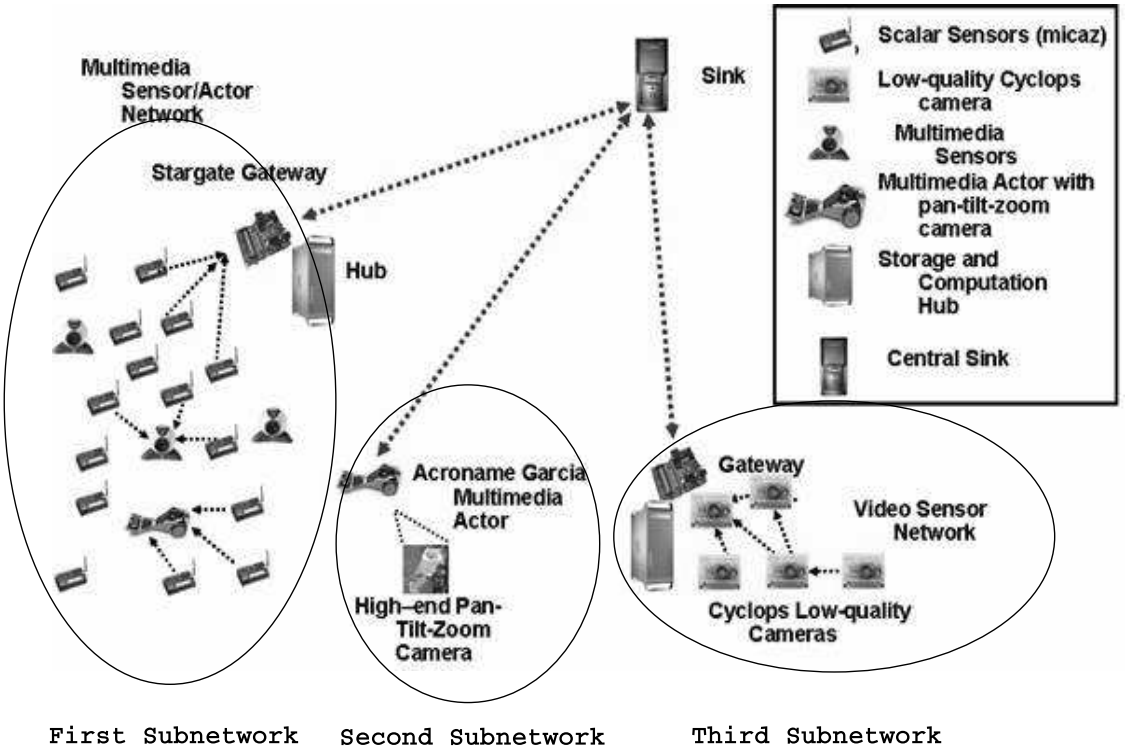


Figure 2.2: Georgia Tech's Testbed for Wireless Multimedia Sensor Networks [2] with subnetworks identified

Medium quality video sensor nodes consist of a USB webcam attached to a Stargate node. High quality video is provided by a video camera attached to a mobile robotic platform.These mobile nodes will move to the an area of interest when they receive information from the scaler sensors.

The multimedia sensor nodes send data to the third layer component, designated as a hub node. The hub node is a more capable node, which may be used for computationally intensive data compression and/or aggregation. The hub node ultimately sends data to the data sink.

The second subnetwork consists only of the sink node and the mobile nodes. This network appears to be included in the architecture diagram primarily to show an alternate data path between the mobile node platform and the data sink.

The final subnetwork most closely resembles the network we describe in this work. It consists

of several sensor nodes, each of which is equipped with a low resolution imaging sensor. Each of the sensor nodes consists of a MICAz or MICA2 mote with a Cyclops camera attached. When data is generated by these nodes, it is routed to a gateway node. The gateway node serves as a hub node, as described previously.

2.3 Adjustable Power Transceivers

Building a real IVWSN that meets the design criteria explored in this paper requires the ability to adjust the transmission power to suit the deployed network. There are two possibilities we need to consider for how the transmission power is configured.

First, we may have a network where the nodes are deployed in a known configuration. When the network configuration is known in advance, the transmission power of each node may be set prior to deployment of the nodes.

Second, we may have a network where the nodes are deployed in a random configuration. In this case, the transmission power required to maintain connectivity and minimize the energy consumed by communication must be computed after deployment.

In this second case, we must make the assumption that each node is able to adjust its transmission power to conform to the deployed network topology. Commercial sensor nodes with this capability are already available. One example is the MICAz from Crossbow Technologies, which has 8 documented power settings available [4]. In the future, we believe cognitive radios will allow further software radio control.

Ideally, we would like to reduce the transmission power if there are intermediate nodes be-

tween the current sending node and the next hop reachable at the current power level. We acknowledge that only a limited number of power levels will be available, so it may not be possible to achieve ideal results in all cases.

In order to maintain connectivity, an algorithm needs to be developed by which a node can adapt its transmission power to communicate with its neighbors. In a regular network, a simplistic algorithm would start at the lowest power setting available and try to connect to the network. Power would be increased, if necessary, until a connection with the rest of the network is established.

Several more advanced power control algorithms have been presented in the existing literature. LINT and LILT, presented in Section 2.1.3.1, approach the problem from the standpoint of maintaining a minimum number of connections to a given node. These algorithms start each node at the highest power level and reduce the power level until the desired connectivity level is achieved. ATPC, described in [24] and B-MAC-PC, described in [25] adapt the transmission protocol to maintain link quality, through Link Quality Indicator (LQI) and/or Received Signal Strength Indicator (RSSI) metrics.

Minimally, the topology control algorithm should execute during the network initialization phase, but it may be run at other times as nodes are added to or removed from the network. We want to stress here that there is no need to frequently invoke the power control process in our design because there is no topology change due to nodes sleeping.

Analytical Study

3.1 Analysis Models

For our analysis, we consider the following network topologies:

1. A 1 dimensional chain of n nodes [3]

2. A 2 dimensional $n \times m$ lattice of nodes [3]

3. A 2 dimensional hexagonal cellular network, that approximates a random network. [26, 3]

3.1.1 A 1 dimensional chain of n nodes

In a 1 dimensional chain of n nodes, the network nodes appear as in Figure 3.1. We add relay nodes to the network to cut the transmission distance in half, as shown in Figure 3.2. This requires the addition of $n - 1$ nodes to the network. We will denote the total nodes in the augmented 1 dimensional chain of nodes as $n' = 2n - 1$. The data will traverse from the node S to the sink D through the chain.

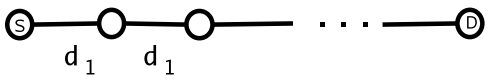


Figure 3.1: 1 dimensional chain of nodes, with distance between nodes of d_1

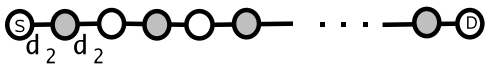


Figure 3.2: 1 dimensional chain of nodes augmented with additional (shaded) nodes such that the distance between nodes is $d_2 = d_1/2$.

3.1.2 A 2 dimensional $n \times m$ lattice of nodes

Our first 2 dimensional model contains n columns and m rows of nodes. This provides a total of $z = n \times m$ nodes in the original network, as illustrated in Figure 3.3. As with the 1 dimensional chain, we are primarily concerned with traffic that flows parallel from left to right or top to bottom in the network. The diagram illustrates this situation, where there are multiple source nodes, marked S, on the left and multiple data sinks, marked D, on the right. We also consider a second case in this network where one of the source nodes is picked randomly as the data source, and one of the destination nodes is picked randomly as the sink.

When we cut the transmission distance in half by adding nodes to the network, as illustrated in Figure 3.4, we have a total of $z' = (2n - 1) \times (2m - 1) \approx 4z$. Notice that we have added approximately $3z$ nodes to the augmented network, i.e. 3 nodes for each node in the original network.

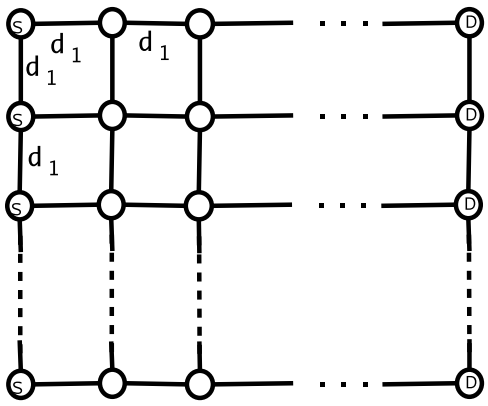


Figure 3.3: An $n \times m$ lattice of nodes, with a distance between nodes of d_1

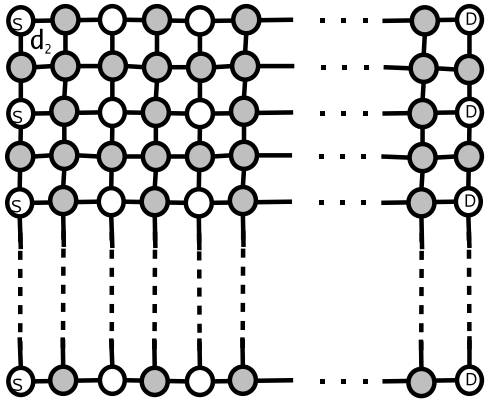


Figure 3.4: An $n \times m$ lattice of nodes augmented with additional (shaded) nodes, such that the distance between nodes is $d_2 = d_1/2$.

3.1.3 A 2 dimensional hexagonal cellular network that approximates a random network

We next examine a 2 dimensional network where the nodes are grouped within the boundaries of a hexagonal cell. While the actual sensor nodes may be randomly distributed within the plane, for purposes of our analysis, we assume that each cell contains at least one node, and that all transmissions emanate from the center of each cell. Each cell has 6 neighboring cells. As argued in

[26], this hexagonal structure also provides a model for calculating maximal interference values in an ad-hoc wireless network, including networks with mobility. Due to this maximal interference property and support for mobile nodes, we believe that the hexagonal model here is a valid approximation for a random sensor network.

We consider a tessellation of the plane by these hexagonal cells. Figure 3.5 illustrates how the augmented hexagonal structure can be built from the original hexagonal network.

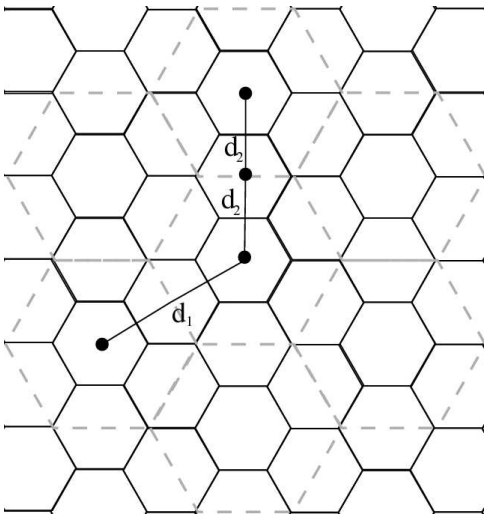


Figure 3.5: A 2 dimensional network of hexagonal cells with z' total cells (solid lines), overlaid with the 2 dimensional network of hexagonal cells with z total cells (dashed lines) covering the same sensor field.

As with our previous lattice of nodes, the original hexagonal network contains a total of z nodes. Similarly, when we cut the transmission distance in half by adding nodes to the network, we need to add 3 new nodes for each of the nodes in the original network, giving a total of $z' = 4z$ nodes in the augmented network.

3.2 Transmission Power Savings With Relay Nodes

We base our power consumption calculations on the RF propagation properties discussed in [27]. We start our analysis by noting that receive power (P_{RX}) is related to transmit power (P_{TX}) with the following equation:

$$P_{RX} \propto P_{TX}(\frac{1}{D})^{\alpha} \tag{3.1}$$

or alternatively:

$$P_{TX} \propto P_{RX} \cdot D^{\alpha}. \tag{3.2}$$

Where α is the path loss exponent and D is the distance at which the transmission is received. Valid values for α in various environments are discussed in [27], and summarized in Table 3.1.

Table 3.1: Path Loss Exponent for Different Environments

Environment	Path Loss Exponent, α
Free space	2
Urban area cellular radio	2.7 to 3.5
Shadowed urban cellular radio	3 to 5
In building line-of-sight	1.6 to 1.8
Obstructed in building	4 to 6
Obstructed in factories	2 to 3

Radio equipment requires some minimum receive power in order to receive a transmission correctly. We assume in this work that the transmission power can be tuned such that received power is always this minimum level, regardless of the distance over which the transmission occurs. As a result, P_{RX} is considered a constant, and transmit power is proportional to the distance over

which the transmission must occur.

$$P_{TX} \propto D^{\alpha}, \alpha \in [2, 6] \tag{3.3}$$

In order to provide a result which is comparable to energy saving methods previously published in the literature, the analysis which follows only takes into account the energy cost associated with communication. We acknowledge that the cost of capturing and processing data in an IVWSN is higher than it may be in other types of sensor networks, thus the energy consumed is not negligible. Therefore, we expect actual energy saving ratio, when considering capturing and processing, to be lower than what our analytical results predict. Our energy saving comparison is presented in Section 3.6.

3.2.1 A 1 dimensional chain of n nodes

In the one dimensional case, we assume the distance between any node and its nearest neighbor node, in the original network, is d_1. This same distance in the augmented network is d_2. As noted in section 3.1, the distance between nodes in the augmented network is half the distance in the original network, therefore

$$d_1 = 2d_2.$$

So, power consumption of a single hop transmission in the original network (P_1) is

$$P_1 \propto d_1^{\alpha}. \tag{3.4}$$

And power consumption of a single hop transmission in the augmented network is

$$P_2 \propto d_2^{\alpha} = (d_1/2)^{\alpha}. \tag{3.5}$$

This gives us a per hop transmission power ratio of

$$P_2/P_1 = ((d_1/2)/(d_1))^{\alpha} = \frac{1}{2^{\alpha}}. \tag{3.6}$$

If we assume $\alpha = 4$, which is the typical case for many sensor application environments, then $P_2/P_1 = 1/16$.

This implies that the augmented network can maintain the same level of connectivity provided by the original network, and at the same time decrease the energy cost of transmission of any given message to each sensor by a factor of 16.

Recall to achieve this result we had to almost double the number of hops for each transmission, therefore the total energy cost of a single message traversing the length of the augmented network is approximately 1/8th the cost of traversing the original network.

3.2.2 A 2 dimensional $n \times m$ lattice of nodes

In the two dimensional case, we again cut our transmission distance in half. Equations 3.4, 3.5, and 3.6 hold for the transmission of a message over a single hop.

In the case of parallel traffic flows, the lattice can be considered a collection of 1 dimensional chains. Therefore, no further analysis is required for determining the energy cost associated with

sending a message.

When the data is sent from a random source node to a random sink node, we need to determine the number of hops required to send the message. in this case, the average number of hops in the original network is

$$h_1 = O(n + m). \tag{3.7}$$

In the case of the augmented network, the average number of hops is

$$h_2 = O(2 \cdot (n + m)) = 2 \cdot O(n + m). \tag{3.8}$$

Since it takes, on average, 2 times as many hops to traverse the augmented network, we conclude the average power cost of sending a message from one sensor node to another in the two dimensional case as being approximately 1/8th the cost of sending the same message across the original network, using $\alpha = 4$, while, again, the energy cost of transmission of any given message to each sensor is reduced by a factor of 16.

3.2.3 A 2 dimensional hexagonal cellular network that approximates a random network

In the two dimensional hexagonal cellular network, we again cut our transmission distance in half. As with the previous network instances, equations 3.4, 3.5, and 3.6 hold for transmissions over a single hop.

Based on the formulas presented in [26], the average number of hops in the original network

is

$$h_1 = O(\sqrt{z}) \tag{3.9}$$

and for the augmented network, the average number of hops is

$$h_2 = O(\sqrt{z'}) = O(\sqrt{4z}) = O(2 \cdot \sqrt{z}) = 2 \times O(\sqrt{z}). \tag{3.10}$$

Since it takes, on average, 2 times as many hops to traverse the augmented network, we conclude the average power cost of sending a message from one sensor to any destination in the augmented hexagonal cellular network as being approximately 1/8th the cost of sending the same message across the original hexagonal cellular network.

3.2.4 Generalizations

If we assume the distance between two nodes in the original network is d_1 and the distance between any two nodes in the augmented network is d_2, we can generalize our results to arbitrary augmented networks. Without loss of generality, we assume that $d_1 \geq d_2$.

If we let h_1 be the number of hops each transmission requires in the original network and h_2 be the number of hops required by a transmission between the same two endpoints in the augmented network, we note

$$\frac{h_1}{h_2} = \frac{d_2}{d_1}. \tag{3.11}$$

If we denote the density as δ, we find

$$\delta \propto \frac{1}{d^2}. \tag{3.12}$$

As a result, the number of nodes can be related to the density by the following equation

$$\frac{n_1}{n_2} \propto \sqrt{\frac{\delta_1}{\delta_2}}. \tag{3.13}$$

If we generalize Equation 3.6, we calculate the ratio of power between a single hop transmission in the original network and the augmented network as

$$P_2/P_1 = (d_2/d_1)^\alpha. \tag{3.14}$$

The end to end energy cost of transmitting a message from one node to any destination in the original network, denoted E_1, is

$$E_1 \propto ((d_1)^\alpha) \times (h_1).$$

For the augmented network, E_2 is

$$E_2 \propto ((d_2)^\alpha) \times (h_2).$$

Therefore, the end to end energy cost ratio between the augmented network and the original network is

$$E_2/E_1 \propto ((d_2/d_1)^{\alpha}) \times (h_2/h_1)$$

$$= ((d_2/d_1)^{\alpha}) \times (d_1/d_2)$$

Thus,

$$E_2/E_1 \propto (d_2/d_1)^{\alpha-1}, \tag{3.15}$$

while the transmission related energy cost at each sensor is decreased in the augmented network by a factor of

$$(d_1/d_2)^{\alpha}. \tag{3.16}$$

3.3 Effects on Network Capacity

The following network capacity considerations are based on the results presented in [3]. In the following discussion, we'll let W represent the channel capacity.

We make the general assumption here that we can augment the network with nodes that act purely as relay nodes, and therefore add no traffic of their own to the network. The reasons we feel this is a valid assumption are discussed in Section 3.5.

Furthermore, for analysis purposes, we will make use of a radio propagation model by which the transmission range and interference range are not the same. In particular, a transmission over a single hop will successfully transmit at a distance of D, but will interfere within other transmissions at a distance of $2 \cdot D$. This phenomenon is shown in Figure 3.6 for a 1-dimensional chain of nodes.

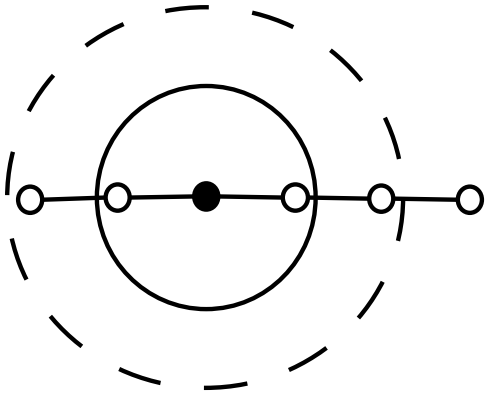


Figure 3.6: Transmission range (solid circle) and interference range (dashed circle) in a one dimensional chain of nodes

Please note that we must maintain the assumptions of section 3.2. This implies that the transmission range and interference range are proportional to the distance between nodes. As a result, when we increase the density, the number of nodes with which any given node interferes is the same in both the original and augmented networks.

3.3.1 A 1 dimensional chain of n nodes

In the case of our 1 dimensional network of nodes, it was shown in [3] that the maximum throughput available to each node in the chain without relay nodes is $W/4$, due to the fact that every node must wait at least 3 transmission cycles to avoid interfering with the next hop nodes' transmissions. In order to achieve this maximum result we assume that there is only one traffic generator in the chain of nodes, as discussed in section 3.1.1.

Our modified network including relay nodes does not affect the ideal bandwidth available per node. We are limited not because of the adjacent sensor nodes, but due to interference with the intermediate relay nodes.

Since our relay nodes do not generate any traffic of their own, our augmented network has the

same number of traffic producing nodes. This implies that the routing load of the original sensors will not increase, despite the increase in node density.

3.3.2 A 2 dimensional $n \times m$ lattice of nodes

In the case of our 2 dimensional lattice of nodes, it was shown in [3] that the ideal throughput available to each node in the network is $W/12$ if all the traffic flows are traversing the network from left to right or top to bottom. This is due to the fact that interference conditions allow only every third row or column of the network to be actively sending data.

In the augmented 2 dimensional lattice of nodes with parallel traffic flows, most of the nodes within interference range of the sender will be nodes added for relay purposes. Due to this behavior, we believe parallel data flows can exist on every other row or column containing sensor nodes, i.e. every other row in the original network. Therefore, we expect throughput of parallel flows to be half that of individual chains of nodes, or $W/8$.

If the traffic pattern is more general, requiring data to flow both horizontally and vertically, [3] shows an expected maximum throughput per flow of approximately $1/2$ of the total throughput of flows which are only moving vertically or horizontally. This ideal rate is based on a theoretical model by which the horizontal and vertical traffic flows are synchronized so that only one direction is actively sending data.

When this more generalized traffic pattern occurs in our augmented lattice, we expect roughly the same effect on throughput as occurs in the original network. A clever routing protocol may be able to take advantage of idle relay nodes to achieve better than expected results.

3.3.3 A 2 dimensional hexagonal cellular network that approximates a random network

For our simplified random network, we have based our capacity analysis on the work presented on the capacity of ad hoc wireless networks in [3].

In the original network with z nodes, we note [3] indicates the following: The total one hop capacity of the network is

$$C_1 = O(z). \tag{3.17}$$

The average number of hops each message may take is

$$h_1 = O(\sqrt{z}). \tag{3.18}$$

The total end-to-end capacity of the network is

$$C_{N1} = O(z)/O(\sqrt{z}) = O(\sqrt{z}). \tag{3.19}$$

Thus, per node throughput is

$$T_{N1} = O(\sqrt{z})/z = O(\frac{1}{\sqrt{z}}). \tag{3.20}$$

Similarly, for our augmented network, which has $4z$ nodes, the total one hop capacity of the network is

$$C_2 = O(4z) \propto 4 \cdot O(z). \tag{3.21}$$

The number of hops each message must take is

$$h_2 = O(\sqrt{4z}) \propto 2 \cdot O(\sqrt{z}). \tag{3.22}$$

The total end-to-end capacity of the network is

$$C_{N2} = (4 \cdot O(z))/(2 \cdot O(\sqrt{z})) \propto 2 \cdot O(\sqrt{z}). \tag{3.23}$$

Assuming all nodes produce traffic, per node throughput is

$$T_2 = (2 \cdot O(\sqrt{z}))/4z = \frac{1}{2} \cdot O(\frac{1}{\sqrt{z}}). \tag{3.24}$$

Since only z nodes in our augmented network are sensor nodes producing traffic of their own, the total per node throughput in our augmented network is

$$T_2 = (2 \cdot O(\sqrt{z}))/z = 2 \cdot O(\frac{1}{\sqrt{z}}). \tag{3.25}$$

As a result of this analysis, we conclude the capacity in our augmented network is approximately double the capacity of the original network.

We also note that similar results can be derived from Equation 11 in [26]. We chose not to present these results here due to assumptions in [26] that force a solution which cannot be easily generalized.

3.3.4 Generalization

To determine generalized capacity formulas for our dense IVWSN deployment, we again turn to the results presented in [3].

In the original network with z nodes, we note [3] indicates the following: The total one hop capacity of the network is

$$C_1 = O(z). \tag{3.26}$$

The number of hops each message must take is

$$h_1 = O(\sqrt{z}). \tag{3.27}$$

The total end-to-end capacity of the network is

$$C_{N1} = O(z)/O(\sqrt{z}) = O(\sqrt{z}). \tag{3.28}$$

Per node throughput is

$$T_1 = O(\sqrt{z})/z = O(\frac{1}{\sqrt{z}}). \tag{3.29}$$

Similarly, for our augmented network, which has $a \cdot z$ nodes in the general case, the total one hop capacity of the network is

$$C_2 = O(a \cdot z) = a \cdot O(z). \tag{3.30}$$

The number of hops each message must take is

$$h_2 = O(\sqrt{a \cdot z}) = \sqrt{a} \cdot O(\sqrt{z}). \tag{3.31}$$

The total end-to-end capacity of the network is

$$C_{N2} = (a \cdot O(z))/(\sqrt{a} \cdot O(\sqrt{z})) = \sqrt{a} \cdot O(\sqrt{z}). \tag{3.32}$$

Assuming all nodes produce traffic, per node throughput is

$$T_2 = (\sqrt{a} \cdot O(\sqrt{z}))/(a \cdot z) = \frac{1}{\sqrt{a}} \cdot O(\frac{1}{\sqrt{z}}). \tag{3.33}$$

However, since only z nodes in our augmented network are sensor nodes producing traffic of their own, the total per node throughput in our augmented network is

$$T_2 = (\sqrt{a} \cdot O(\sqrt{z}))/z = \sqrt{a} \cdot O(\frac{1}{\sqrt{z}}). \tag{3.34}$$

As a result, we conclude that the augmented network has an increased capacity of approximately $\sqrt{a}$.

3.4 Latency

In many sensor network applications, particularly surveillance, command, and control applications, there is a requirement to get the data back to the data sink within a time limit, so that it may be

used for prompt decision making. We intended to study the impact of dense deployment on the ability to deliver data within a time limit.

As illustrated in Figure 3.7, we need to consider two separate, but closely related, factors when discussing latency in wireless sensor networks. First, we need to consider the physical distance over which the transmission occurs. Second, but more importantly, we need to consider the number of hops over which the transmission occurs.

Although the geographic transmission distance remains the same between the source and sink nodes, the transmission distance may change depending on the selected route between the source and destination. In particular, the energy savings methods explored in Section 2.1 only consider maintaining connectivity when determining if a node is allowed to sleep. No consideration is made to the length of the path required to transmit a message from the source to the destination. As a result, this potentially increases the physical distance over which the transmission occurs. Thus, this increase in distance translates instantly into an increase in transmission related latency.

More significantly, although the routing delay may vary for each message relayed through a node, we expect routing delays proportional to the number of hops the transmission takes. Primarily, this is because routing in wireless sensor networks is accomplished using store and forward methods.

In the case of the power saving methods explored in Section 2.1, where the nodes are allowed to turn off their communication hardware, a significant amount of delay per hop may be incurred. With these methods, it is possible for the next-hop node in the communication path to be either in the sleeping state, or transitioning between the sleeping and active states. If the next hop node is inactive, and there is no alternate path available, there is an additional holding delay incurred, e.g.,

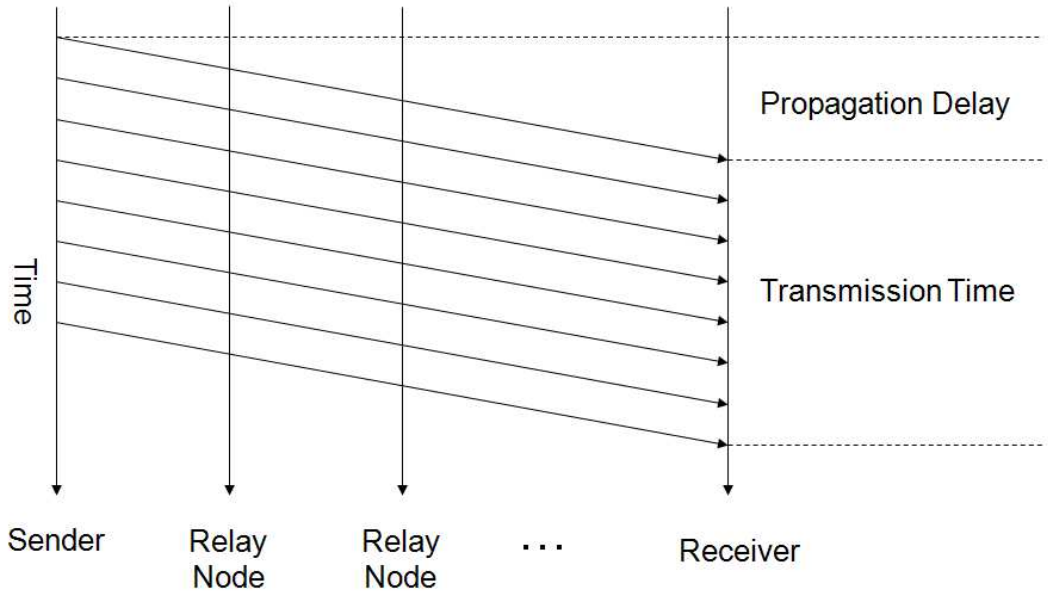


Figure 3.7: The factors involved in determining latency: Propagation Delay, including any routing delays, and transmission time.

on a MICAz mote, this transition can take up to $1.3ms$ [5]. The collective delay added by inactive next hop nodes makes these algorithms a poor choice for real time applications (e.g., video sensor networks). Consider a connection with $n = 10$ transmission hops. Using MICAz motes, the delay incurred due to transmission from the sleeping to the active state may be as long as 13ms.

In our density derived method, the aggregate physical distance over the multi-hop path between any given source and destination pair in the original network changes very little in the augmented network. Furthermore, since we don't let the communication hardware sleep on any node, the path between source and destination nodes is static, so long as all nodes still have power available. Thus, there will be little deviation in the holding delay induced by each node in a given path during the lifetime of the network. As a result of these observations, we concern ourselves primarily with latency induced by the number of hops in the network, when assessing the delay in a dense deployment of sensors.

Since all multi-hop routing can be reduced to a chain of transmissions passing through intermediate relay nodes, we will examine the effect of increased density on transmission latency using the model of a 1 dimensional chain of n nodes . In this model, we will assume that the time it takes to send a packet between two nodes in a single hop is T_1, which includes both propagation and routing delays, and that this time is constant regardless of the distance of that hop or the routing load of the nodes involved.

As a result, for a single transmission, the time it takes to traverse the hops between the source node and the destination node is

$$T_{single} = (n-1) \times T_1 \tag{3.35}$$

However, since each image in our transmission consists of multiple packets, Equation 3.35 only reflects the propagation and routing delays inherent with sending the data over multiple hops between the two points. The total transmission time for an image must include the time it takes to send all packets associated with the image. We assume each image has to be packed into M packets, so must therefore consider the time it takes for all M packets of the image to be delivered. Our interference model from Section 3.3, indicates the sender must wait 3 transmission cycles between each packet, so the total time it takes to send an image is given by,

$$T^0_{total} = (n-1) \times T_1 + (M-1) \times 3 \times T_1 = (n + 3 \times M - 4) \times T_1 \tag{3.36}$$

in the original network and

$$T^1_{total} = (2 \times n - 2) \times T_1 + (M-1) \times 3 \times T_1 = (2 \times n + 3 \times M - 5) \times T_1 \tag{3.37}$$

in the augmented network with 1 node between each of the original nodes.

Now, if we insert k additional nodes between each pair of original nodes, the augmented network will consist of a total of $k \times (n-1) + n$ nodes, thus the total transmission time is

$$T_{total}^{k} = [(k+1) \times (n-1) + 3 \times (M-1)] \times T_1 \tag{3.38}$$

Let us now consider how additional hops of our dense deployment will effect transmissions in an IVWSN deployment. We will consider an original network such that $n = 10$, for example.

The image size we capture with a Cyclops camera is 128x128 pixels at 3 bytes per pixel, or 49152 bytes. The maximum packet size we can send using a MICAz mote is 128 bytes. If we assume 108 bytes of the packet is available for sending the payload,$M = \lceil 49152 bytes / 108 bytes \rceil = 456$.

In this scenario,

$$T_{total}^{0} = (10 + 3 \times 456 - 4) \times T_1 = 1374 \times T_1 \tag{3.39}$$

in the original network and

$$T_{total}^{1} = (2 \times 10 + 3 \times 456 - 5) \times T_1 = 1383 \times T_1 \tag{3.40}$$

in the augmented network.

As a result, for a 1-dimensional chain with a relatively small value of n, doubling the number of hops does not significantly increase the latency for delivering the entire image.

However, equations 3.36-3.38 also imply that as the number of hops ($n-1$ in the case of Equation 3.36) increases, the routing delay induced by the number of hops, h, becomes dominant in the latency calculation. If T_{max} is the maximum delivery time allowed for a given application, then

$$(h + 3 \times M - 3) \times T_1 \leq T_{max} \tag{3.41}$$

and

$$h \leq \frac{T_{max}}{T_1} - 3 \times M + 3 \tag{3.42}$$

Consider the augmented network with k nodes inserted between each pair of original nodes. Since $h = (k+1) \times (n-1)$, Equation 3.42 also imposes an upper limit on the value of k. In other words, the latency requirement will impose an upper limit on the density of the network.

This leads us to the conclusion that while we theoretically could deploy a network with arbitrarily high density for the purpose of power saving, delay constraints provide an upper bound on what network density can effectively perform the task at hand. Please note, there is also a lower bound on network density, due to network connectivity and sensor coverage requirements [9, 19].

3.5 Cost Considerations

In our analysis, we have primarily concerned ourselves with sensor nodes for which the cost of the sensor component exceeds the cost of the communication and processing components, e.g., an image sensor module may cost more than the communication module. In this situation, in order

to achieve the results presented thus far, it may be desirable to build the augmented network using nodes without sensors.

These sensorless nodes are to be dedicated to functioning as relay nodes. Ideally, these dedicated relay nodes should provide the same communication and computational facilities provided by the sensor carrying nodes.

In this situation a net cost saving is made in the initial purchase of equipment with a trade-off of reduced sensor redundancy in the network.

3.6 Comparison to Existing Energy Saving Methods

In this section we will look at how our method stands up against previous power saving methods described in the literature. Table 3.2 provides a comparison of network lifetimes provided by the algorithms presented in Section 2.1. In order to provide comparable results, the numbers reflected in the table are the multiples of a base network lifetime for which the network will effectively survive according to theoretical and/or simulated results provided in the references. When multiple results are available, the larger result was used. Additionally, as noted previously, none of the results presented take into account the cost of gathering the data, which we believe will be a significant energy consumer during the lifetime of an IVWSN.

From an energy perspective, the expected performance from our algorithm appears to be considerably better than the majority of the competing ideas presented thus far.

The exception to this is when STEM is used in combination with GAF. The dramatic energy savings possible with STEM is a result of the sparse network traffic assumption made by the

authors of [11], which we believe does not always apply to our multimedia sensor networks.

In the interest of completeness, we must acknowledge that all of the methods cited which allow nodes to sleep may provide further energy savings when the deployment density is increased. Unfortunately, these methods still suffer from the expense of decreased capacity and increased latency as outlined in section 2.1.

Table 3.2: Network lifetime in terms of base network lifetime

Method/Algorithm	Theoretical	Simulated
SPAN [21]	3.5	2
Abbas [14]		1.4
GAF [13]		6
BECA [12]		1.2
AFECA [12]		1.55
STEM [11]	2	
STEM + GAF [11]		100
Iranli [15]		3
Density (our work)	16	

Experimental Study:Development of an Image/Video Sensor Network Testbed

4.1 Preliminary Simulation Results

At this point in time, we have not conducted any simulations of our own. As a starting point for our own simulations, we turn to [3]. In this paper, the authors present simulation results verifying the capacity limits of a 1 dimensional chain of nodes and a 2 dimensional lattice of nodes, which we have used as the basis of our own analysis.

The simulation results presented in [3] are based on an Ad Hoc network consisting of nodes using IEEE 802.11 for the communication channel, with a data rate of 2Mbps. The nodes are able to correctly receive transmissions when separated by a distance of 250 meters, but can interfere at a distance of 550 meters.

For the simulations, the nodes were separated by a distance of 200 meters, and each transmitted packet was 1500 bytes.

As a baseline simulation, the authors created a network with only two nodes. They found

the nodes were able to communicate at a rate of 1.7Mbps, despite the 2Mbps setting of the simulated radios. The reason for this discrepancy is the overhead involved with the 802.11 protocol itself. The required CTS,RTS, and ACK packets consume the 0.3Mbps not available for user data communication.

As we presented in Section 3.3, we expect a chain of nodes to be able to provide a throughput of 1/4 the maximum throughput of a single hop transmission. This would indicate that the expected throughput of the simulated chain of nodes should be $1/4 \times 1.7Mbps = 0.425Mbps$. These expected values correspond with the expected values presented in [3], however, the simulations indicate the throughput achieved approaches 0.25Mbps, which is only 1/7 of the maximum value. The simulated throughput results are presented in Figure 4.1.

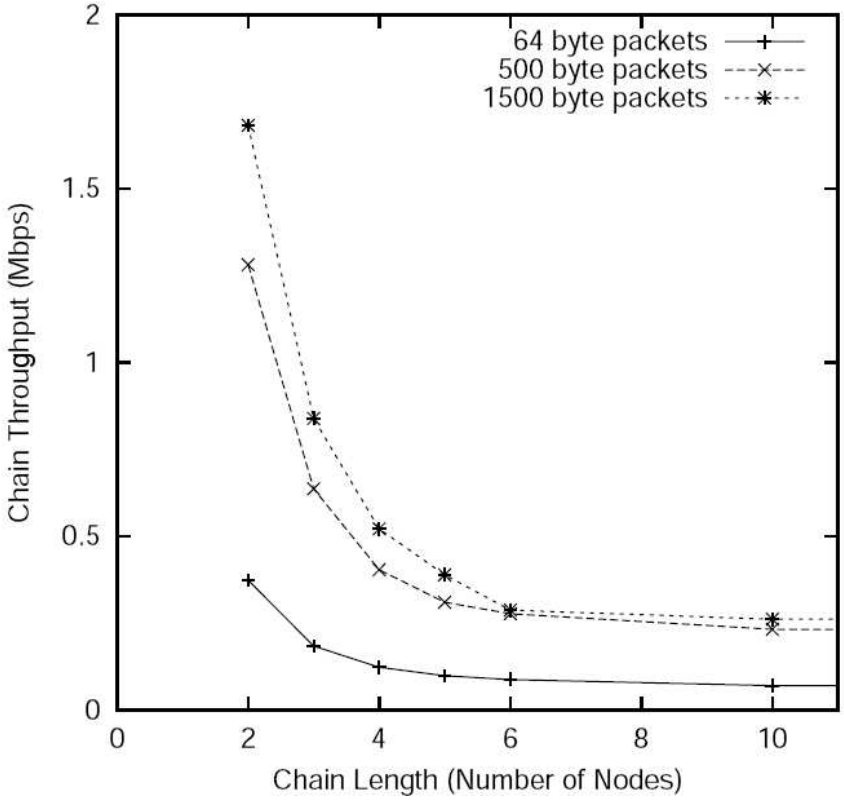


Figure 4.1: Throughput for a chain of nodes as a function of the length of the chain [3]

The authors explain this discrepancy through additional simulations. They discovered that if the packets were paced by the node, rather than by the 802.11 protocol, a maximum rate of 0.41Mbps could be achieved. The conclusion drawn is that 802.11 can transmit at nearly the predicted rate, but it is unable to determine the appropriate schedule of transmissions on its own.

We also presented in Section 3.3 the expected throughput of a lattice of nodes with parallel traffic flows to be 1/12 the maximum value. This implies the simulated value should be $1/12 \times 1.7Mbps = 0.14Mbps$. Again, these expected values correspond with the expected values presented in [3].

The simulated network is a square lattice, i.e., the number of nodes in each chain is equal to the number of chains in the lattice. In this network framework, the simulated results show the per flow throughput for 1500 byte packets settles at a value of around 0.1Mbps, as depicted in 4.2. The authors use the previously discussed discrepancies of the 802.11 protocol to explain the difference between the simulated value and the expected value.

For a set of communication in which the flows are not parallel, we expect a maximum throughput of 1/2 the parallel flow rate, or 1/24 of the the maximum flow. This indicates an expected value of $1/24 \times 1.7Mbps = 0.071Mbps$. The simulation results for this traffic pattern indicate a maximum throughput of approximately 0.045Mbps, as shown in Figure 4.3. The authors explain this discrepancy by noting that 802.11 detects collisions and then enters a backoff period before allowing a second attempt at retransmission. Successive collisions increase the length of time spent backing off, causing a reduction in the actual number of concurrent transmissions.

These simulation results indicate that our theoretical capacity limits for the 1 dimensional chain of nodes and the 2 dimensional lattice of nodes are sound, though not achievable when using

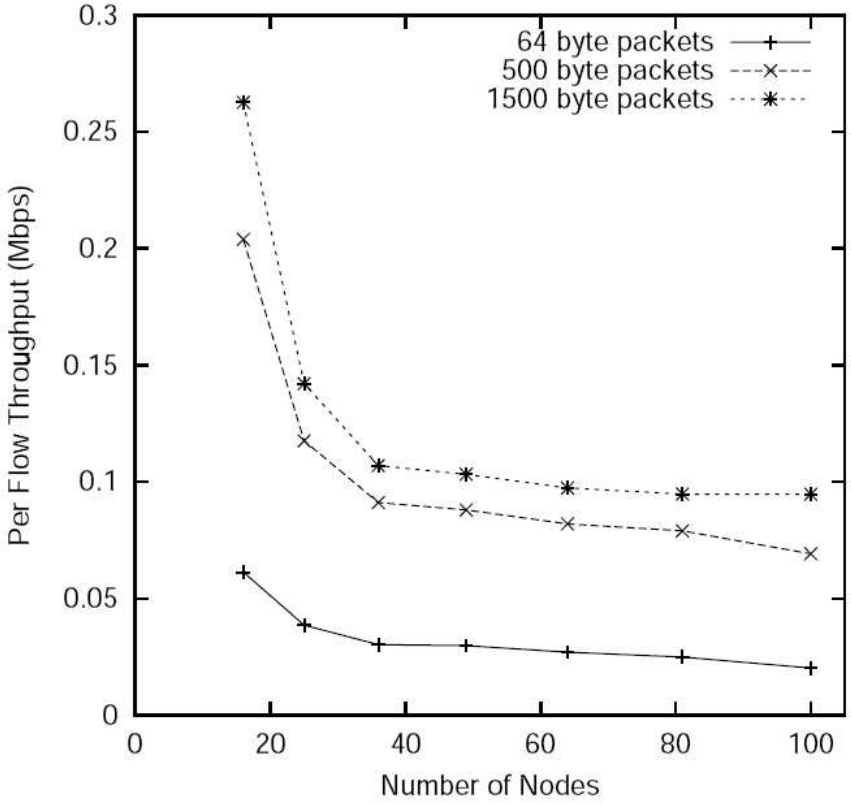


Figure 4.2: Per flow throughput for a lattice of nodes with parallel traffic flows as a function of network size [3]

IEEE 802.11 as the Mac layer protocol. It is still necessary to determine how increasing the density of the network affects latency and network lifetime.

4.2 A Multimedia Testbed Structure

As a major part of our experimental study, we have built a testbed for multimedia sensor networks. This work is conducted as part of an existing NSF funded project for building a sensor network testbed at Wright State University.

Our existing sensor network consists of a set of Crossbow Technologies MICAz motes connecting to a data sink via a wireless communication link. For purposes of implementing our multi-

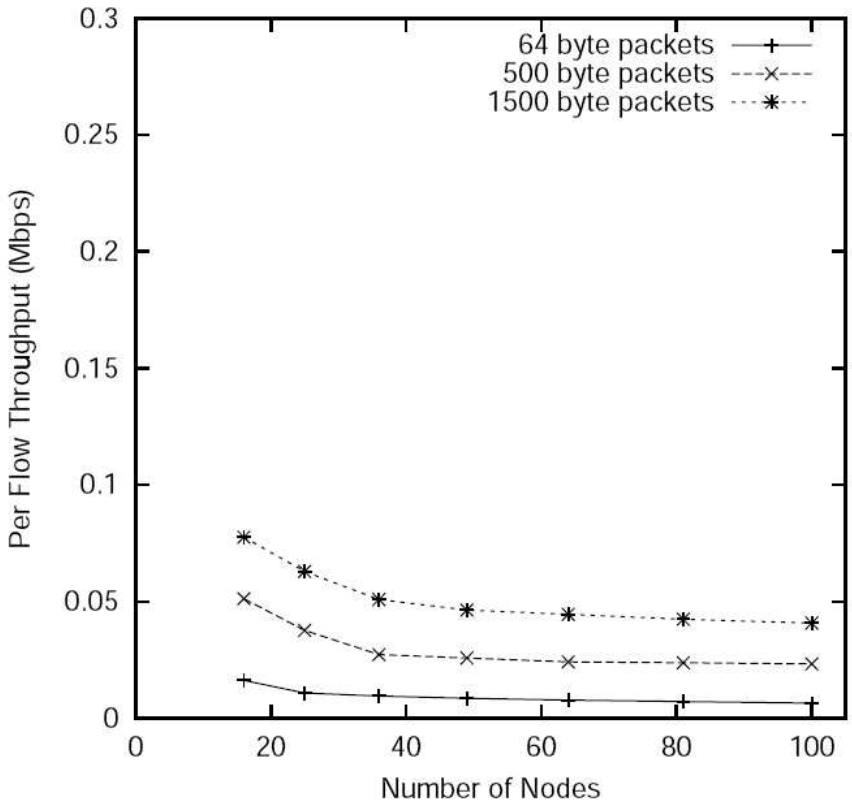


Figure 4.3: Per flow throughput for a lattice of nodes with crossing traffic flows as a function of network size [3]

media sensor network, we have added Cyclops cameras manufactured by Agilent Technologies [7]. The Cyclops cameras can be attached directly to the existing sensor interface port on the MICAz.

Ideally, the sensor network we setup will have randomly placed multimedia sensors in the sensor field. This ideal situation is depicted in Figure 4.4. Also shown in the figure are wireless communication links between the nodes in the sensor network, and between the sensor network and Stargate nodes acting as WiFi gateways.

The WiFi connections are used to transmit data to the data sink, which may be a portable device such as a PDA or a laptop computer. For testing purposes, the connection to the data sink may be transmitted over a wired connection from the node acting as the sensor network gateway to a laptop computer.

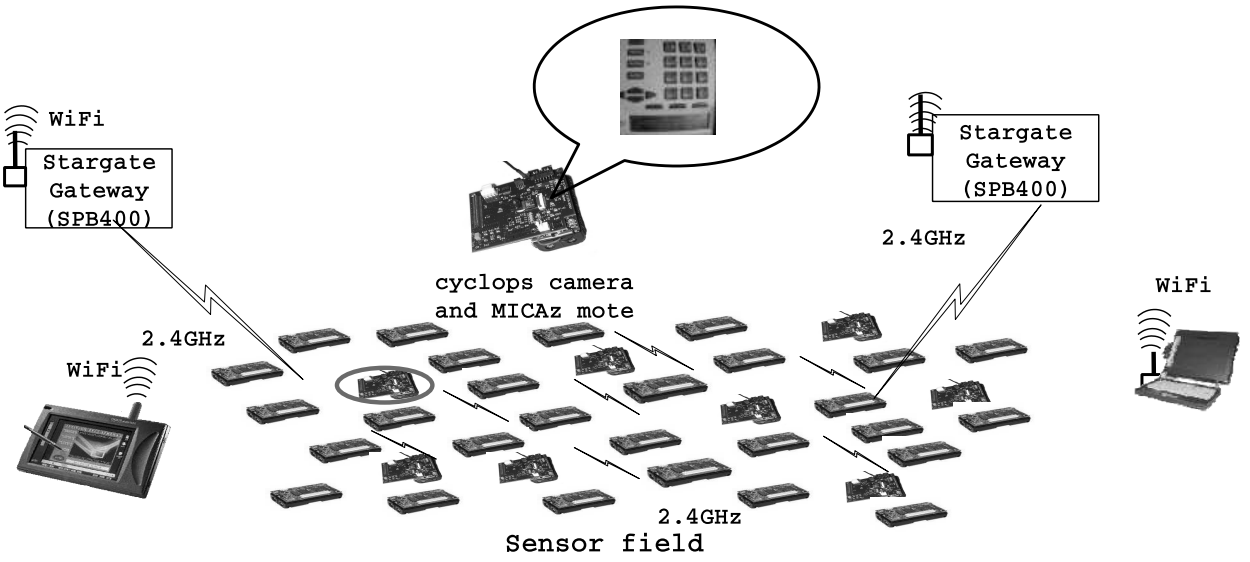


Figure 4.4: Multimedia Sensor Network Overview.

We have conducted tests to verify the effects of density changes on energy consumption, capacity, and latency by deploying the test network in configurations conforming to the regular network structures presented analytically in prior sections of this work.

4.2.1 Crossbow Technologies MICAz

The Sensor node we utilize for our experiments is the Crossbow Technologies MICAz mote. This device consists of a processor board and a holder for two AA batteries. The processor board contains an Atmel ATmega 128L microcontroller and a Chipcon CC2420 Radio Transceiver. The CC2420 is an IEEE 802.15.4 compliant radio transceiver. For connection to other devices, the processor board includes a 51 pin high density connector [4]. A MICAz is shown in Figure 4.5.

Figure 4.5: A Crossbow Technologies MICAz mote

4.2.2 Agilent Technologies Cyclops Camera

The Agilent Technologies Cyclops Camera is the sensor module we utilize for our experiments. It was developed by Agilent in conjunction with the Center for Embedded Network Sensing at UCLA. The Cyclops Camera consists of a single board with an Atmel ATMega128L microcontroller connected to an Agilent Technologies ADCM-1700 camera module through an Xilinx XC2C256 CPLD module [7]. A Cyclops Camera is shown in Figure 4.6.

4.2.3 Crossbow Technologies Stargate Gateway

The Crossbow Technologies Stargate Gateway is a single board computer based on an Intel StrongArm Processor. The Stargate runs the Linux Operating system. Because the Stargate Gateway is

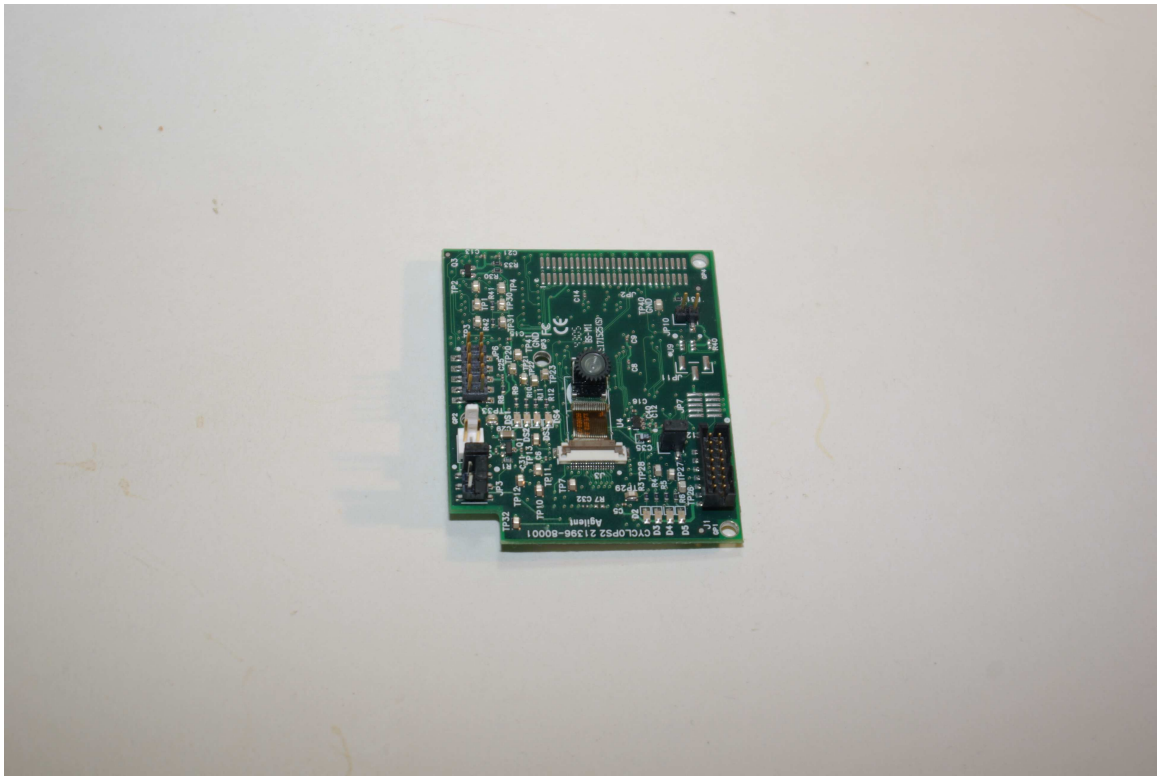


Figure 4.6: An Agilent Technologies Cyclops Camera

a more powerful device than the MICAz motes, it can be distributed in the network to act as a data collection point and bridge between the sensor field and other networks, such as the Internet.

For connection to external devices, the Stargate contains a PCMCIA slot, a Compact Flash slot, an RS232 Serial Port, an Ethernet port, a USB port, and a 51 pin high density connector which can be connected to a MICAz mote [23]. A Stargate Gateway is shown in Figure 4.7.

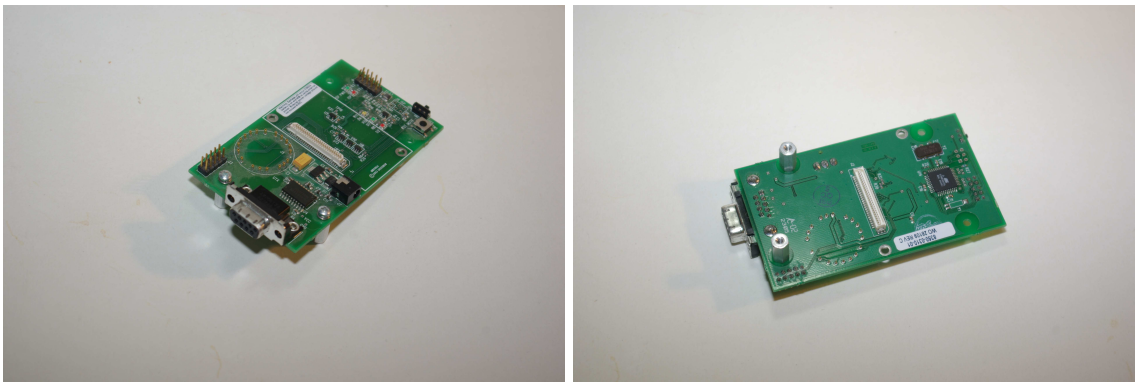
4.2.4 Crossbow Technologies MIB510

The Crossbow Technologies MIB510 is a programming board used to connect a MICAz mote to a PC. The MIB510 features two 51pin connectors and and an RS232 serial port. One of the two 51pin connectors is identical to the connector on the MICAz motes. This connector is used

(a) top (b) bottom

Figure 4.7: A Crossbow Technologies Stargate Gateway

for programming the Cyclops Cameras. The second connector is identical to the connector on the Cyclops Camera and other sensors modules. This connector is utilized for programming and communication with the MICAz mote from the PC. The MIB510 physically connects one of the RS232 serial ports on the MICAz mote or Cyclops Camera to a 9 pin D-sub connector for use with the PC [28]. An MIB510 is shown in Figure 4.8.



(a) top (b) bottom

Figure 4.8: A Crossbow Technologies MIB510

4.3 A Representative Experimental Set up for Image Sensor Systems

All of our experimental studies started with using the release code for the cyclops cameras. This code was written for Crossbow Technologies MICA2 motes, but we are using MICAz motes, which communicate via a different physical layer, requiring some modification of the code. We will briefly examine the flow of control for each of the three communication paths involved, and discuss how we had to change the source code to accommodate the difference between the MICA2 and MICAz.

The communication occurs in three phases, as illustrated in Figure 4.9. First, we have communication between the Cyclops camera and the MICAz mote to which it is attached, as illustrated in Figure 4.10. This communication occurs via the I2C bus. The program running on the Cyclops camera, *captureI2Ctest*, is responsible for capturing the image and then building the payload of the message that is eventually sent out via the radio. The program running on the mote, called *MoteI2CRelay*, is responsible for triggering the image capture and then sending out each of the packets making up the image as shown in Figure 4.11.

The request made by the MICAz mote running *MoteI2CRelay* to the Cyclops camera running *captureI2Ctest* includes information on what size and type of image the camera should capture. The cyclops camera is capable of delivering 3 types of images in 3 different resolutions. The image types available are 1 bit per pixel Black and White images, 1 byte per pixel greyscale images, and 3 byte per pixel RGB color images. The available image sizes are 32x32 pixels 64x64 pixels and 128x128 pixels. The size of images captured with each possible pair of parameters is shown in

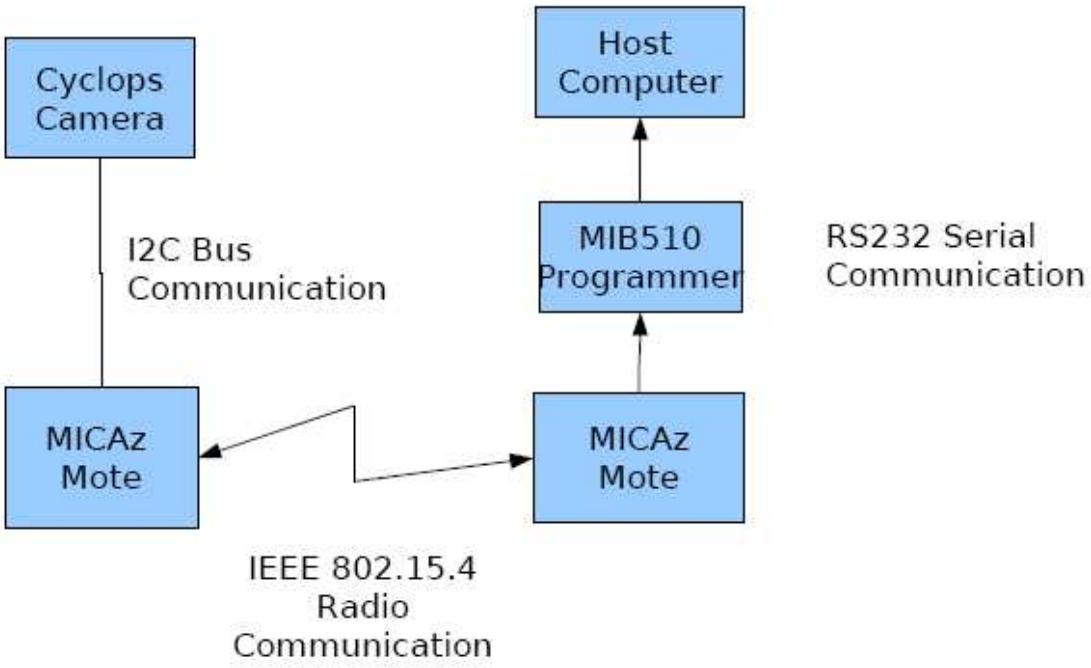


Figure 4.9: The Flow of Data from Cyclops Camera to Host PC via MICAz motes.

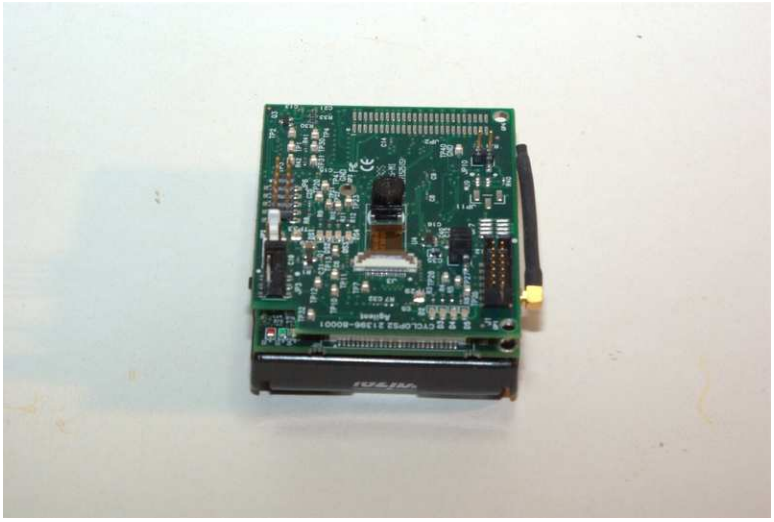


Figure 4.10: A Cyclops Camera attached to a MICAz Mote.

Table 4.1.

Next, we have communication between two MICAz motes, one attached to the Cyclops and a second attached to the host computer. This communication occurs via an IEEE 802.15.4 compliant

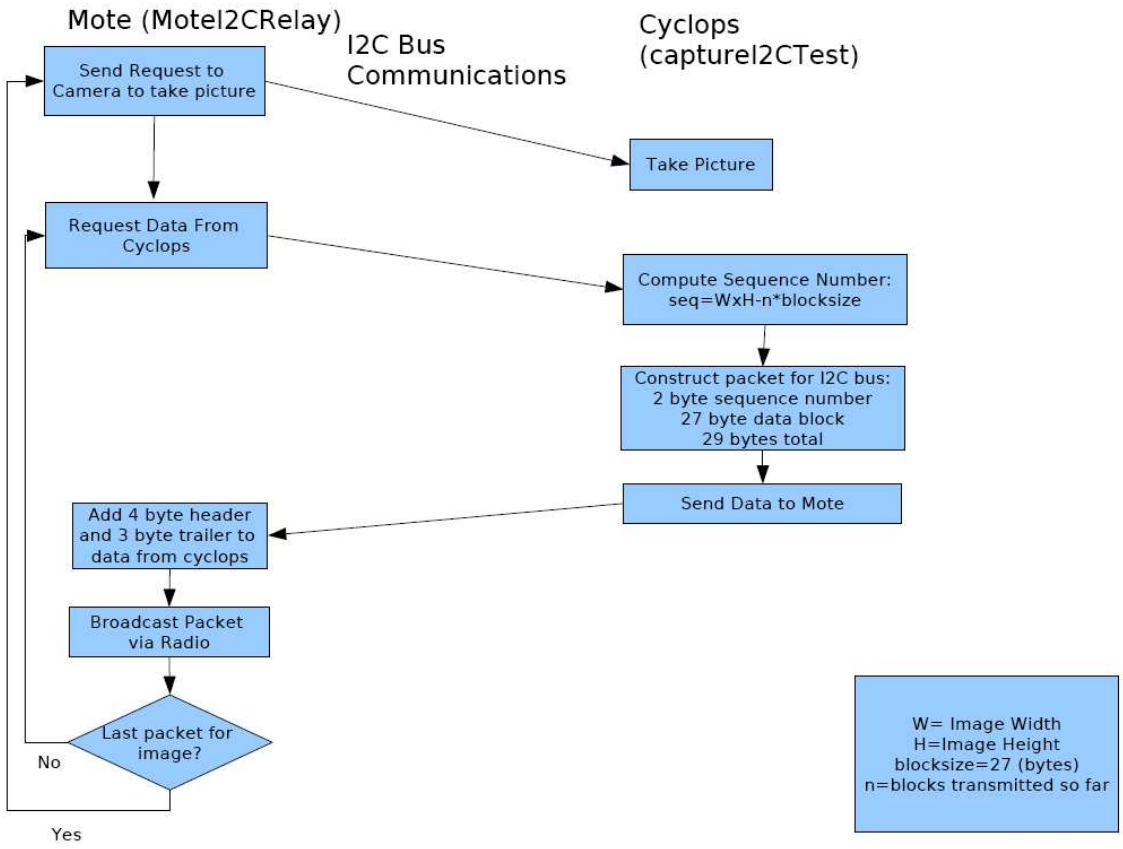


Figure 4.11: The Flow of Data from Cyclops Camera to Host PC via MICAz motes.

Table 4.1: Total Image Size based on Cyclops parameters

	32x32 pixels	64x64 pixels	128x128 pixels
1 bit per pixel	128 bytes	512 bytes	2048 bytes
1 byte per pixel	1024 bytes	4096 bytes	16384 bytes
3 bytes per pixel	3072 bytes	12288 bytes	49152 bytes

radio stack, based on a Chipcon CC2420 radio transceiver.

Finally, we have the communication between the MICAz mote acting as the base radio and the host computer. On the host computer, which is either a PC attached to the mote via a serial programming board or a Crossbow Technologies Stargate Gateway connected directly to a MICAz mote via a 51 pin connector.

The flow of control between the MICAz mote acting as the base and the host computer is

depicted in Figure 4.12 for a PC based host using a Crossbow MIB510 as an interface board. A MICAz Mote Connected to an MIB510 is shown in Figure 4.13. The data is transfered from the MICAz mote, running a program called *GenericBase*, to a program on the host computer called *frame*. The *frame* program reads the packets from the serial port and reconstructs the image. Error detection on the radio link is handled by the MICAz mote. The *frame* program only verifies that the sequence number indicates the data is within the limits of the image.

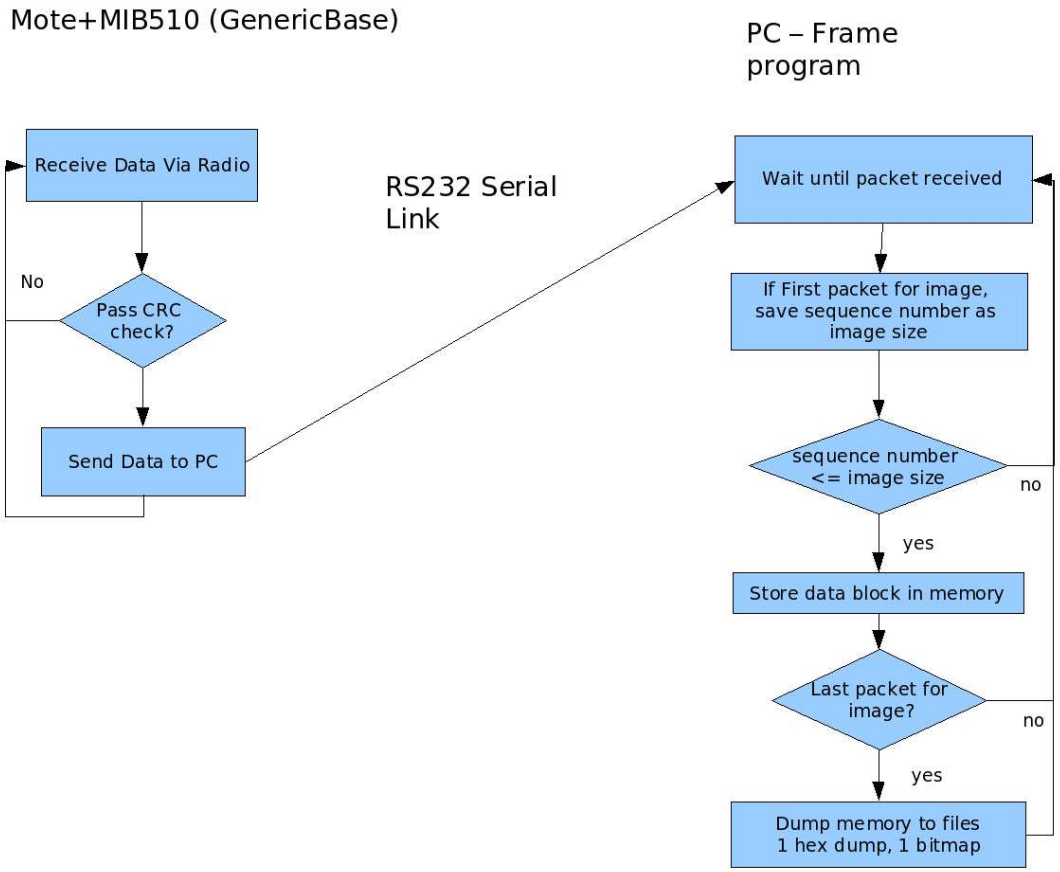


Figure 4.12: The Flow of Control from the MICAz Mote to the host computer.

As an alternative to using a PC and an MIB510 to connect to a MICAz mote, we have the option of utilizing a MICAz mote attached to a Crossbow Technologies Stargate Gateway, as seen in Figure 4.14. The Stargate Gateway is a single board computer based on an Intel StrongArm

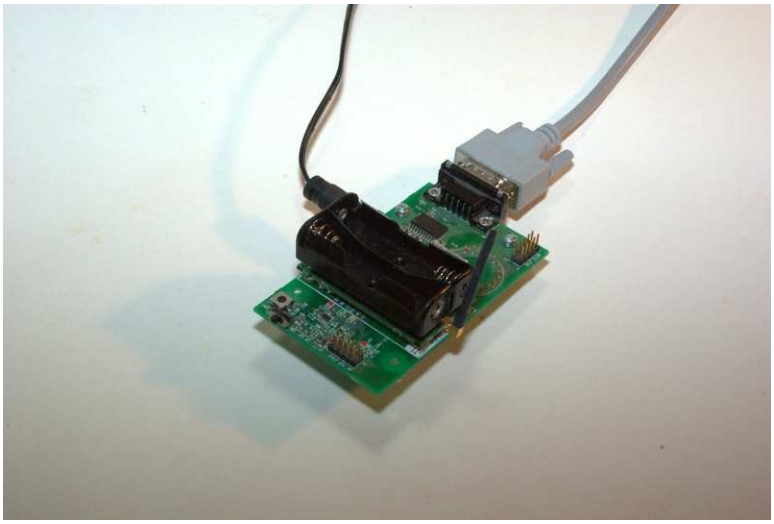


Figure 4.13: A MICAz Mote attached to a Crossbow MIB510, which is used as an interface to the Sensor Network for a PC.

Processor. The Stargate runs the Linux Operating system [23].

Figure 4.14: A MICAz Mote attached to a Crossbow STARGATE Gateway, which is used as an interface to the Sensor Network for a PC.

In addition to a direct connection to a MICAz mote, a Stargate has several other advantages over a PC based interface platform. First, the Stargate can be used as a gateway between the sensor field and the Internet through an Ethernet connection. This makes it possible to access the sensor field through a web server running on the Stargate Gateways. This allows monitoring several

sensor fields from a centralized location via the Internet, as depicted in Figure 4.15. We refer to this configuration of sensor nodes as a Sensor Web. The flow of data within the Sensor Web is depicted in the block diagram shown in Figure 4.16.

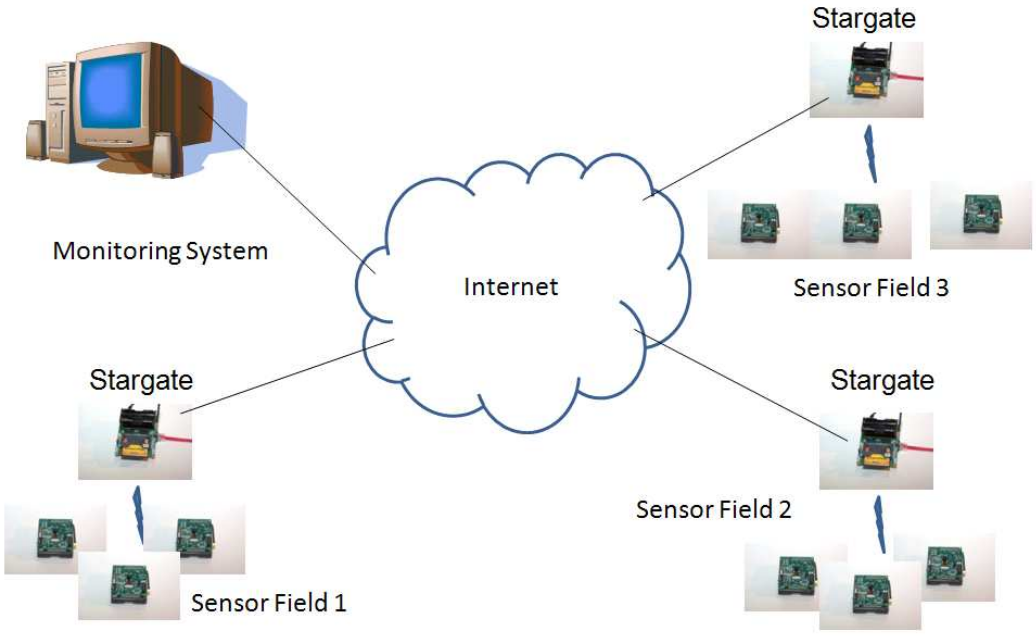


Figure 4.15: Using Stargate Gateways to interface multiple Sensor Fields via the Internet, into a Sensor Web.

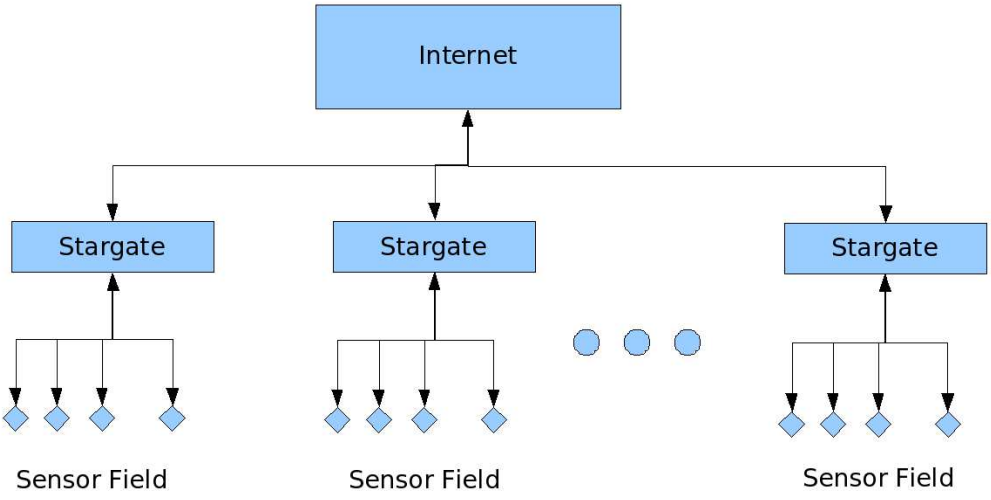


Figure 4.16: Block Diagram showing components of a Sensor Web and the data flows between each component.

Second, a Stargate Gateway can be configured via an EXT2 formated Compact Flash device to interface with the Sensor Network. On startup, the Stargate is configured to read commands from a BASH shell script in a file on the Compact Flash device called *cfcard.rc*. This allows setting up a Stargate to record data for later retrieval without any network connection, where the Stargate node serves as a more capable higher tier sensor node, as depicted in Figure 4.4. Third, this allows rapid deployment of several identical systems.

We must point out at this point that the packet structure of the MICA2 and MICAz motes differs due to the different radio modules used. Specifically, the byte order is different, resulting in the message payload starting at a different point within the MICAz message than it did in the MICA2 message. In order to accommodate this difference in structure, we modified the *frame* program in the PC to support MICAz motes. The data packet formats are depicted in Figure 4.17.

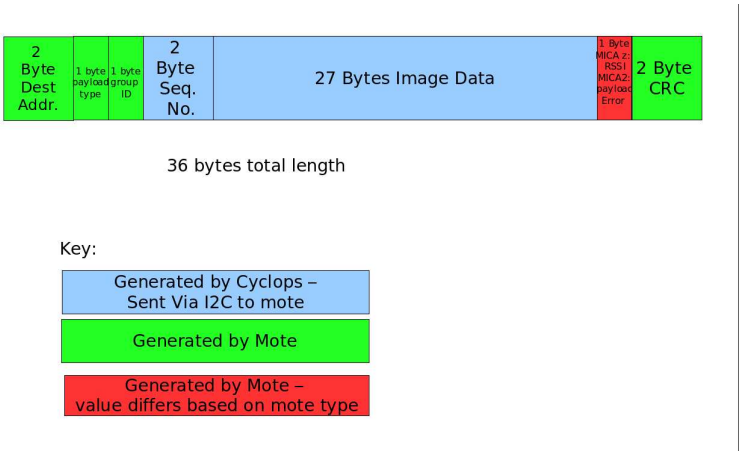


Figure 4.17: Data format received by the host computer.

4.4 Experimental Study on the Camera Power Consumption

To validate our claim in the preceding chapter that capturing and processing an image may take more energy than transmitting that image, we designed and ran two series of single hop experiments.

In the first series of experiments, we attached a cyclops camera to a MICAz mote and set the camera to take 128x128 pixel greyscale images. This mote was configured to capture an image, send the image out via the radio, and wait a specified amount of time before repeating the process. We ran this experiment with a 60 second wait time. In this configuration, it takes 3 seconds for the image to transmit, so the total cycle time is 63 seconds, as illustrated in Figure 4.18. The source code for the program loaded to the mote appears in Appendix B.

A second mote was attached to either a PC via an MIB510 programming board or directly to the 51 pin connector on a Stargate Gateway. This mote was configured to run the standard tinyOS *GenericBase* program, which forwards all received packets through a serial port to the host computer. The PC and Stargate were configured to run a C program which reconstructs the received image and records time stamp and signal strength information from each data packet received. The *GenericBase* program appears in Appendix B. The C Program used for this test appears in Appendix A. A sample image from this test is shown in Figure 4.19.

In the second series of experiments, we removed the cyclops from the sending mote and built a new program, *MoteNoDataRelay*, which generates an image with a constant greyscale value instead of getting a real image from the Cyclops. This series of experiments were otherwise identical to the first series. The flow of control in this program is depicted in Figure 4.20. The program code

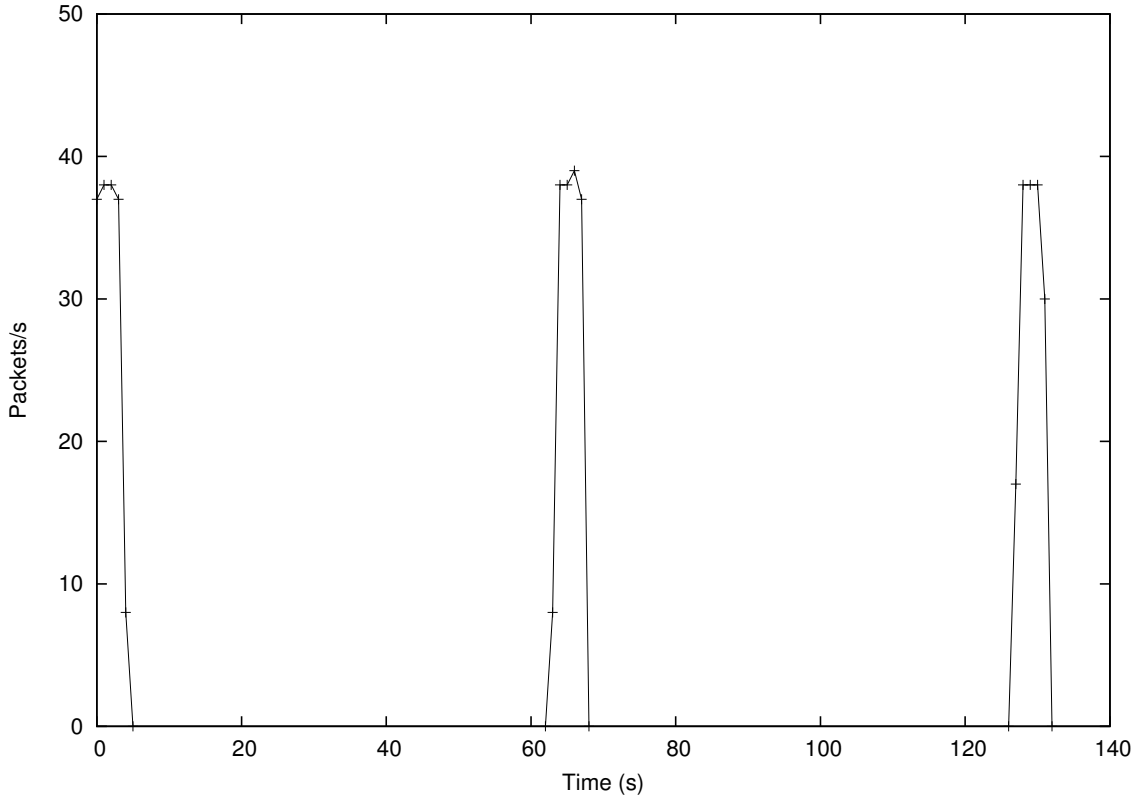


Figure 4.18: Packets/Second for the first 140 seconds of a typical experimental run, with the interval between images set to 60 seconds.

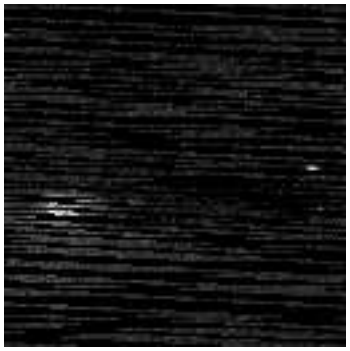


Figure 4.19: Sample image from baseline test with camera

for the sending mote appears in Appendix C. A sample received image from this test can be seen in Figure 4.21.

In both series of experiments, the sensor boards are powered by using brand new Energizer type E91 AA batteries.

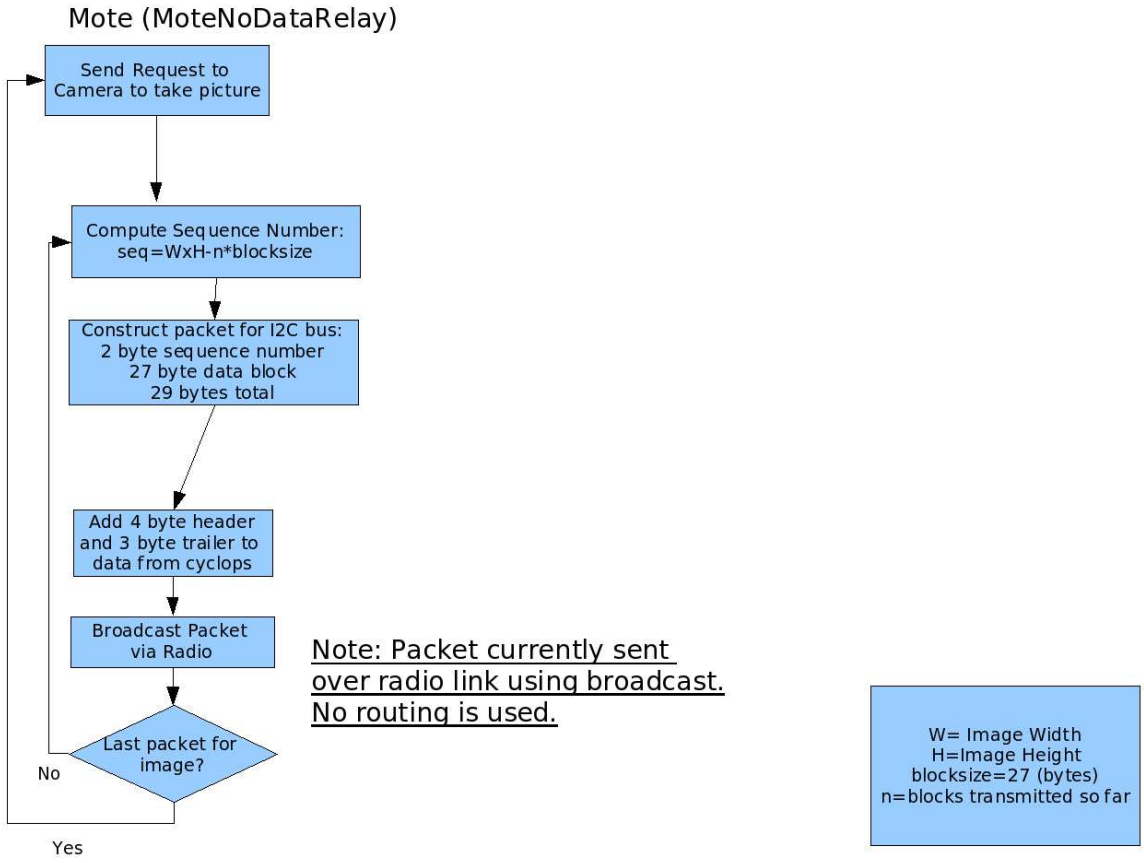


Figure 4.20: The Flow of Control via the MICAz Mote for the constant greyscale Images.

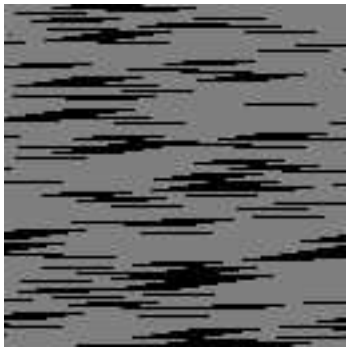


Figure 4.21: Sample image from baseline test without camera

Table 4.2 summarizes the results of both experimental series.

Without examining the available current draw numbers on the data sheets, this result gives a clear indication that the camera consumes a significant amount of the power during the life of the experiment. The data sheets for the sensor module, Cyclops camera, and the CPU and

Trial	Camera Attached	Time (hours)
1	Yes	42.06
2	Yes	41.61
3	Yes	41.35
4	No	100.26
5	No	99.52
6	No	99.40

Table 4.2: Experimental results - 60 second interval

communication module, MICAz mote, indicate that both devices consume $8mA$ of current when they are on but not performing any other tasks, i.e. while the camera is not taking a picture and the mote is not sending data, which is the case during most of the run time in the experiment. As a result, sensing and communication consume a similar percentage of energy. It should be noted: during a period of 3 seconds when the transceiver is transmitting, the CC2420 draws a higher current of $17.4mA$. Similarly, a $15mA$ current is drawn when the image sensor is in the process of capturing an image. However, due to the relatively small percentage of time over which the image capture and transmission occurs, this only accounts for a small energy consumption [4, 7].

While this test was not designed to show image quality, we make note at this point of the relatively poor image quality observed in Figures 4.19 and 4.21. The reasons for this poor image quality will be discussed in more detail in Section 4.8.

4.5 MICAz CC2420 Transceiver Power Capabilities

Our experimental study requires a platform capable of adjusting its transmission power. The MICAz motes we are using for our experiments have 8 distinct transmission power levels available. Changing the power level requires changing the Power Register setting of the CC2420 transceiver

built into the MICAz. The 8 available Power Register settings, along with the corresponding current draw and output power are summarized in Table 4.3 [4, 5].

Table 4.3: MICAz transceiver power settings and current draws [4, 5]

Mode	Power Register	RF Power (dBm)	RF Power (mW)	Current Draw (mA)
TX	31	0	1.000	17.4
TX	27	-1	0.7943	16.5
TX	23	-3	0.5012	15.2
TX	19	-5	0.3162	13.9
TX	15	-7	0.1995	12.5
TX	11	-10	0.1000	11.2
TX	7	-15	0.0316	9.9
TX	3	-25	0.0032	8.5
RX				19.7
PD				0.002
Idle				0.426

The CC2420 has 3 additional modes available. In Powered Down (PD) mode, the CC2420's crystal oscillator and voltage regulator are both turned off, and the CC2420 is unable to send or receive any transmissions. In Idle mode, the crystal oscillator and voltage regulator are turned on, and the CC2420 can send and receive transmissions. In receive mode, the CC2420 is in the process of receiving data. Unlike the transmission modes, these three modes have a single fixed current draw available. The current draw for each of these modes is shown in Table 4.3.

Our series of transmission power adjustment experiments utilizes the transmission power setting capabilities of the CC2420. These experiments required a modification to the source code for the *MoteI2CRelay* and *MoteNoDataRelay* programs to allow adjusting the power level. Appendices E and F contain the respective code for each setup.

4.6 Dependency Between Transmission Power and Distance

One of the assumptions we made in our analysis is that if the transmission power decreases by half, then the effective distance over which that transmission may occur must also be reduced by half. In order to determine if this is the case, we setup a simple test using the same software described in Section 4.5.

For this experiment, we laid out 7 points in a straight line at distances of 1,3,5,10,15,20, and 30 meters from the base station. We then set up a sending MICAz mote with each of the 8 possible transmission register values and recorded the Receive Signal Strength Indicator (RSSI) as computed by the receiving MICAz mote at each of the distances possible. The values recorded are listed in Table 4.4

Table 4.4: RSSI Values at given distances

Power Register	RF Power (dBm)	RF Power (mW)	Current Draw (mA)	20m RSSI	15m RSSI	10m RSSI	5m RSSI	3m RSSI	1m RSSI
31	0	1.000	17.4	-	-83	-73	-70	-68	-66
27	-1	0.7943	16.5	-	-84	-74	-68	-70	-69
23	-3	0.5012	15.2	-	-85	-77	-73	-71	-64
19	-5	0.3162	13.9	-	-85	-78	-75	-74	-72
15	-7	0.1995	12.5	-	-	-80	-77	-77	-75
11	-10	0.1000	11.2	-	-	-83	-80	-79	-77
7	-15	0.0316	9.9	-	-	-85	-84	-84	-82
3	-25	0.0032	8.5	-	-	-	-	-	-

We make note especially of two measurements shown in the table, the values for 10m at a power level of 31 and the value for 5m at power level 23. The test resulted in similar RSSI readings for these two cases. These values are significant because the power output of power level 31 is 1mW and the power output for power level 23 is 0.5012mW, so this is an indication that our assumption, holds, the distance at which a transmission will be received will be cut in half if the

output power is reduced by half, holds.

4.7 Adjustable Transmission Power and its Effect on Single Hop Transmission

To start determining how much power savings we can obtain by reducing the transmission distance in a real world situation, we conducted a set of single hop experiments to observe the effect of changing the transmission power. We conducted these tests utilizing the software described in Section 4.5.

In the first series of experiments, we set the MICAz's transmission power to its highest setting, 31, which transmits at 0 dBm or 1 mW. As with the previous series of experiments, we generated a constant greyscale image which was transmitted with a defined wait time between each image. We ran this experiment with two wait times, 5 seconds and 60 seconds. As with previous tests, the images take 3 seconds to transmit in this configuration. As a result, the cycle time with a 60 second wait is 63 seconds, and with a 5 second wait is 8 seconds.

In the second series of experiments, we set the transmission power to 11, which is -10dBm or 0.1mW. This series of experiments was otherwise identical to the first series.

Table 4.5 summarizes the results of the 60 second interval experiments and Table 4.6 summarizes the results of the 5 second interval experiments.

Table 4.5: Experimental results - 60 second interval

Trial	Power Level Setting	Time (hours)
1	31	100.26
2	31	99.52
3	31	99.40
4	11	99.20
5	11	105.97
6	11	99.87

Table 4.6: Experimental results - 5 second interval

Trial	Power Level Setting	Time (hours)
1	31	99.86
2	31	98.35
3	31	104.99
4	11	101.68
5	11	100.89
6	11	101.69

4.7.1 Analysis of Results

While our experiments show a trend towards an improvement, on the order of 3%, we note here that the experimental results are both inconsistent in showing power improvment and the result is not a highly significant demonstration of the power savings we wish to achieve. We expected to see a much more significiant improvement in the length of time when reducing the transmission power by as much as we did.

In an attempt to determine why we see the results we did, we have done some mathematical analysis on using the power consumption numbers from the data sheets and tools provided to us by the manufacturer of the MICAz motes.

To start this analysis, we note that the aggregate amperage consumption of a MICAz mote that is transmitting image data is:

$$C_{total} = (C_{CPU} + C_{camera} + C_{transceiver}) \tag{4.1}$$

Where C_{CPU} is the current draw of the CPU board, C_{camera} is the current draw of the camera and $C_{transceiver}$ is the current draw of the transceiver due to transmission and reception of data.

C_{CPU} is constant, because we never turn the CPU off, but $C_{transceiver}$ and C_{camera} are dependent on the percentage of time the device is in a particular state. The data sheet for the MICAz mote indicates that $C_{CPU} = 8mA$ [4].

$C_{transceiver}$ is composed of three parts, one defining the percentage of time the transceiver is transmitting, one defining the percentage of time the transceiver is receiving data, and the third defining the percentage of time the transceiver is idle. The power consumed during transmission depends on the transmission power. The possible values are defined in Table 4.3. From the table, we can see the power consumed when the transceiver is receiving data is $19.7mA$ and the power consumed when the transceiver is idle is $426\mu A$ [4, 5].

In other words,

$$C_{transceiver} = C_{transmit} \times \frac{T_{transmit}}{T_{cycle}} + C_{receive} \times \frac{T_{receive}}{T_{cycle}} + C_{idle} \times (1 - \frac{T_{receive} + T_{transmit}}{T_{cycle}}). \tag{4.2}$$

C_{camera} is composed of two parts, the current draw while the camera is taking an image and the current draw while the camera is on but not taking an image. The current consumed while the camera is taking an image is $15mA$, and while the camera is on but not taking an image it is $8mA$

[7]. In other words,

$$C_{camera} = C_{capture} \times \frac{T_{capture}}{T_{cycle}} + C_{camera_on} \times (1 - \frac{T_{capture}}{T_{cycle}}). \tag{4.3}$$

Bringing the components together, we find:

$$\begin{aligned} C_{total} =& (8mA) + (7mA) \times \frac{T_{capture}}{T_{cycle}} + 8mA + C_{transmit} \times \frac{T_{transmit}}{T_{cycle}} \\ &+ 19.7mA \times \frac{T_{receive}}{T_{cycle}} + 426\mu A \times (1 - \frac{T_{receive} + T_{transmit}}{T_{cycle}}) \end{aligned} \tag{4.4}$$

We can utilize Equation 4.4 to predict the length of time the batteries should last based on the current capacity of the batteries, nominally $1500mAh$ for the AA batteries powering a MICAz mote. Table 4.7 shows the numerical results our equation predicts for several input parameters.

Table 4.7: Power calculations based on Equation 4.4

Power Level	Wait Time (s)	Camera (Yes/No)	$C_{transmit}$ (mA)	$T_{transmit}$ (s)	$T_{receive}$ (s)	$T_{capture}$ (s)	T_{cycle} (s)	C_{total} (mA)	$\frac{1500mAh}{C_{total}}$ (h)
31	60	Yes	17.4	3	0	0.01	63.01	17.23	87.03
11	60	Yes	11.2	3	0	0.01	63.01	16.82	89.18
31	5	yes	17.4	3	0	0.01	8.01	22.78	65.86
11	5	yes	11.2	3	0	0.01	8.01	19.52	76.86
31	60	no	17.4	3	0	0	63	9.24	162.40
11	60	no	11.2	3	0	0	63	8.82	170.03
31	5	no	17.4	3	0	0	8	14.79	101.40
11	5	no	11.2	3	0	0	8	11.53	130.09

This numerical analysis, based on data sheet information, shows that if we are not receiving data, and we should not be receiving any data from other sensor nodes in this instance, then we should see considerably longer sensor longevities than we observe in our experiments. We believe

this is due to the inability to provide a clean experimental space. In this case, we cannot control if, or how frequently, the transceiver enables its receive mode. We can model our experimental results using Equation 4.4, but only if we include some time with the transceiver in receive mode.

Our current experimental platform does not allow changing the receive power of the transceiver. Because our transmission duty cycle is low, the transceiver module has the potential to operate in receive mode for the majority of time it is active.

4.8 An Energy Efficient Multi-Hop Sensor Network

With the single hop experiments we have completed to this point, we have established that the assumptions we made in our analysis are sound in principle, and that our analytical results hold, at least to some extent, in practice on a single hop.

To complete our work, we now move on to constructing a multi-hop testbed for Image/Video transmission. In starting this work, we need to first point out that the data received by our nodes in single hop experiments, as shown previously in this work, has a lot of missing data. This missing data is due in large part to overflowing the receive buffer on the receiving node. In the case of our MICAz motes, the CC2420 Transceiver has a receive buffer of limited size, 128 bytes, which is capable of holding approximately 4 packets [5]. If we overflow this buffer, packets are dropped. This leads to a requirement of implement flow control and retransmission of packets.

4.8.1 Flow Control

We considered the requirement for flow control with our capacity analysis, but the need for a reliable transport mechanism was not readily apparent until work was begun with the MICAz motes and cyclops cameras.

A simple flow control scheme, pacing the rate of outgoing packets, is capable of preventing most buffer overflows. We have implemented this scheme in our packet transmission code. An image resulting from a simple flow control algorithm is shown in Figure 4.22.

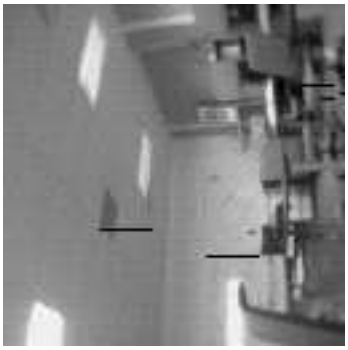


Figure 4.22: Sample image from camera with inter-packet delay set to 10000 μs

The difficult task when implementing a packet pacing scheme is deciding how long to wait between packets. If the pace is too fast, the pacing scheme fails to provide the required inter-packet spacing. If the pace is too long, there is a risk that the data from one image sensor reading will not be sent out before the next scheduled image. The first case is not acceptable, because it allows packets to be dropped by the receiver, which is what we are attempting to avoid. The second case is not acceptable because we are trying to maintain a specified data rate. Although we don't attempt to find the optimal operating point, we find 10000 μs is a reasonable value. The image shown in Figure 4.22 above was generated with a 10,000 μs delay. In a real world deployment, we want the delay to be dynamically allocated, so that it adapts to the transmission requirements of the

deployed network, and the processing speed at each node.

We note here that flow control does not not prevent packet collisions due to contention, it only gives the next hop node a better chance to process and/or retransmit the data before we send the following packet. Figure 4.23 shows an image transmitted over two hops with the flow control mechanism described in this section.

Figure 4.23: An image received via two hops using flow control, but no retransmission of data.

The image in Figure 4.23, there have been several packets lost during the network transfer. In a multihop network, each hop can contribute to loss of packets. The image above was sent by the originating MICAz mote using 820 data packets. The MICAz mote acting as a relay received and retransmitted 819 packets, but the MICAz mote acting as the sink only received 817 of those packets.

In order to minimize loss of packets at each hop, and increase the chance of sending all packets through the network, a more complex reliable transfer scheme is required. This scheme should make use both of packet pacing and acknowledgment that a packet has been received before the next packet will be sent.

Retransmission of packets is generally completed after monitoring for the acknowledgment for a specified period of time and resending a duplicate packet if no acknowledgment arrives within

that time period or a negative acknowledgment arrives during that time period. We have not implemented a retransmission scheme of our own, but we utilize in our experiments the retransmission mechanism provided by the Crossbow *ReliableRoute* module distributed as part of the tinyOS 1.x contributions repository.

4.8.2 Multi-hop Routing

The Crossbow *ReliableRoute* module, is one of several routing protocols provided as part of the tinyOS 1.x which all provide common interface. We have modified the single hop *MoteI2CRelay* and *GenericBase* programs to allow using any of the available routing protocols simply by linking to the specified routing protocol implementation. The source code changes required to add the Crossbow *ReliableRoute* for sensor nodes deployed as relay only nodes or with a Cyclops Camera appears in Appendix G. Updated code for the base mote connected to the host computer appears in Appendix H.

The use of the Crossbow *ReliableRoute* module added an additional 7 bytes of header information to the data sent over the network. This required a change to the *frame* program residing on the host computer. It also reduced the number of payload bytes which can be carried by each frame. The changes to the *frame* program to support the multi-hop format appear in Appendix I. The data format used in a multi-hop packet is described in Figure 4.24.

When sending data via a multi-hop path, a multi-hop send function is invoked instead of the broadcast send illustrated in Figure 4.11. The Multi-hop send in *ReliableRoute* includes a retransmission mechanism as illustrated in Figure 4.25.

In all of the routing protocols implemented for tinyOS 1.x, the destination for packets is the

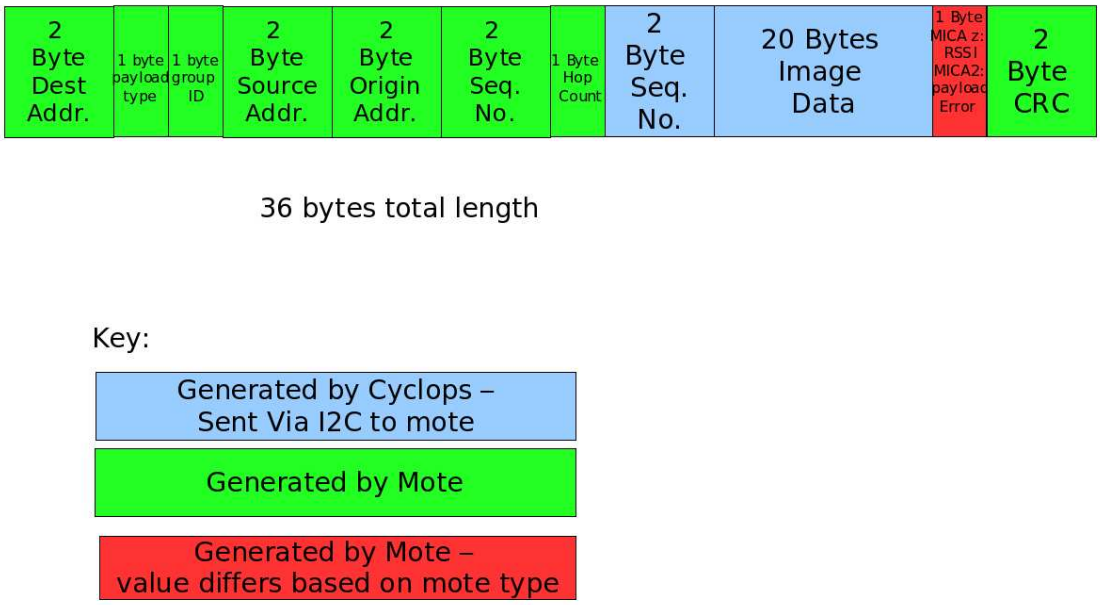


Figure 4.24: Data format received by the host computer.

node with address 0. When a node other than the base receives a packet addressed to it, the packet is forwarded using the process shown in Figure 4.27. Due to the limited lab space for the multi-hop testbed, we have forced the next hop selection to be 1 less than the current address, i.e. we specify a fixed forwarding path as illustrated in Figure 4.26.

The address information for each node is generated by setting a specified location within the executable image loaded on each MICAz mote. Within code running on tinyOS, this address is refereed to as *TOS_LOCAL_ADDRESS*. Setting the address is accomplished by running the *set-mote-id* shell script distributed with tinyOS, or by specifying the id at compile time.

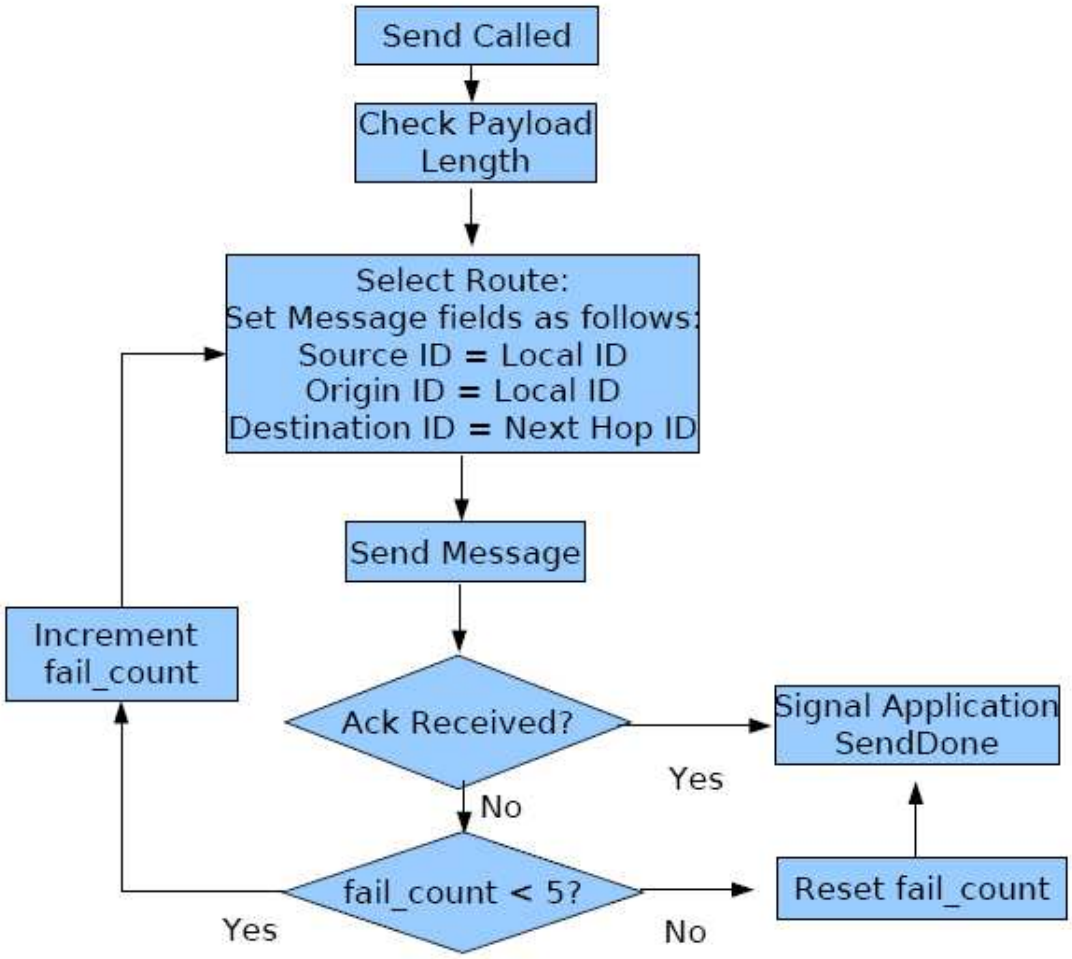


Figure 4.25: Flow chart of the Multi-Hop Send process.

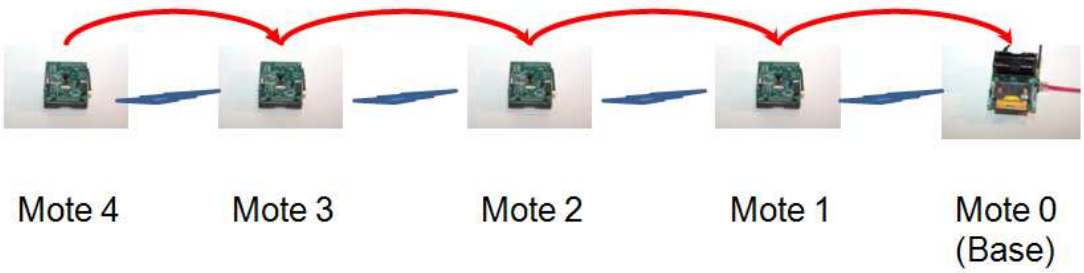


Figure 4.26: Multi-Hop Routing as implemented for testing purposes.

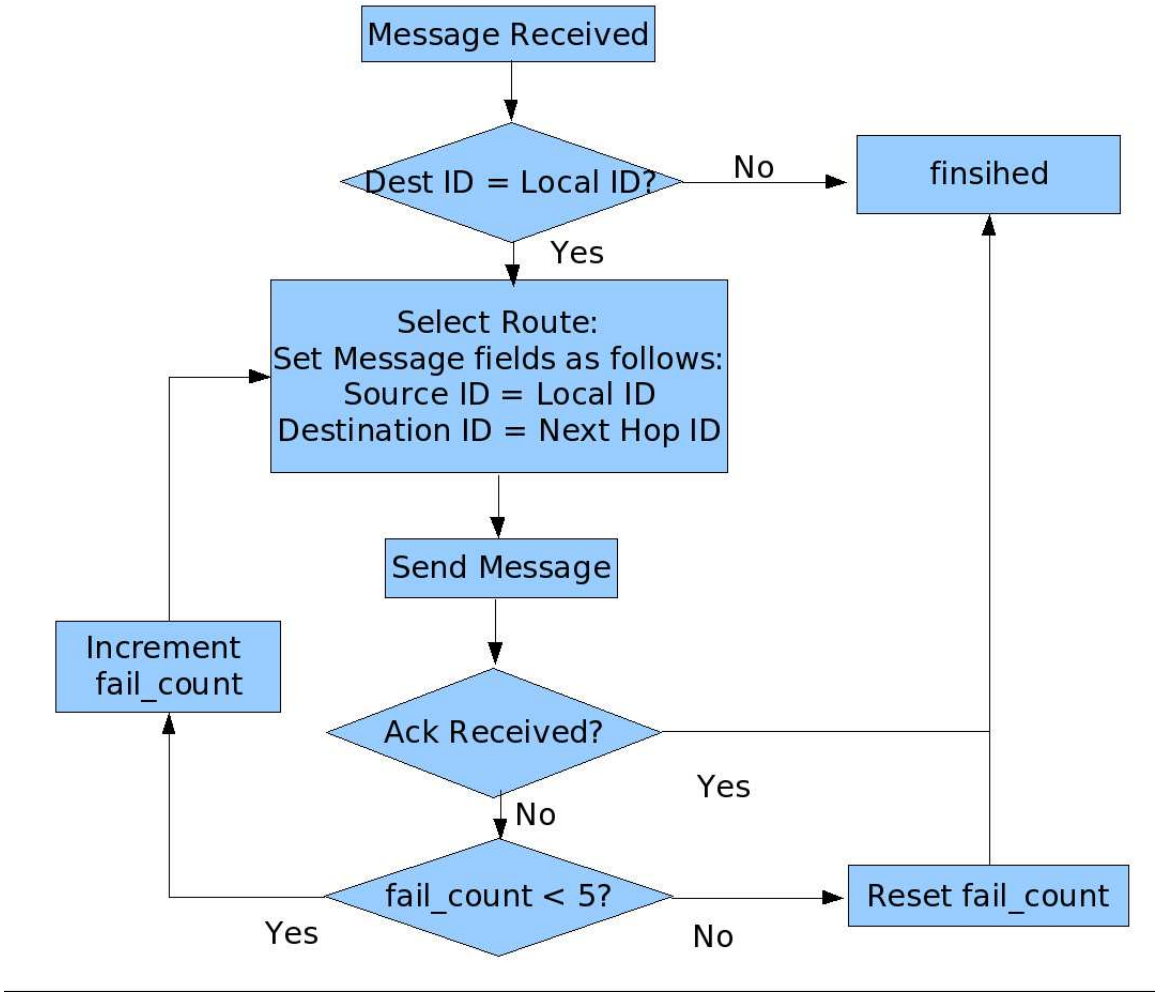


Figure 4.27: Flow chart of the Multi-Hop Forward process.

4.8.3 How Retransmission affects Power Consumption

To provide an end-to-end reliable transport mechanism, we can approach the problem in two general ways.

We can build an end-to-end reliable system where only the endpoints are involved with handling acknowledgments. An advantage to this system is that only the end points need to concern themselves with acknowledgments for any given message. A disadvantage is that if a transmission must be repeated, then every node in the network must retransmit the data.

To minimize the number of retransmissions involved, we can implement an end-to-end reliable system by providing hop-by-hop reliable forwarding. In hop-by-hop reliable forwarding, each pair of nodes handle acknowledgments and retransmissions between the two nodes. This has the advantage of only retransmitting data, if it was not acknowledged, over a single link, not over the whole path.

The hop-by-hop reliable route and endpoint only reliable route methods have different effects on the energy consumption of the nodes in the network. In the end-to-end scenario, when the source node fails to receive an acknowledgment, it has no way to know where in the process the failure occurred. The failure may be due to the packet not reaching the sink node or due to the acknowledgment not reaching the source. In either case, the entire sequence must be repeated to ensure delivery of the packet.

In the hop-by-hop scenario, the source node does not need to worry about retransmission of the packet once the next hop node has taken possession of the packet and acknowledged receipt. Should a failure occur, only the nodes involved in the failure are required to retransmit the data.

We have chosen to use the *ReliableRoute* module because it provides a hop-by-hop reliable transport mechanism.

To further simplify our multihop deployment, we integrated the code required for the base node, the node with address 0 to forward data on to the serial port into the *MoteI2CRelay* program. Included in this integration was an additional queue to allow the relatively slow speed serial port to keep up with the higher speed radio transmissions. The final code for the *MoteI2CRelay* program appears in Appendix J.

Our multihop experiments were conducted on a network consisted of two hops using 3 nodes:

A MICAz mote equipped with a Cyclops Camera, a MICAz mote acting as a relay only node, and a MICAz mote attached to an MIB510 connected to a PC. The sink node received data for approximately 22 seconds for each image, as depicted in Figure 4.28. An example of the images received during this test is shown in Figure 4.29. We note here one minor difference between the multihop experiments and the single hop experiments. The timer which controls when each image starts is started after the image is transfered in the single hop case, but in the multihop case it is started before the image is captured.

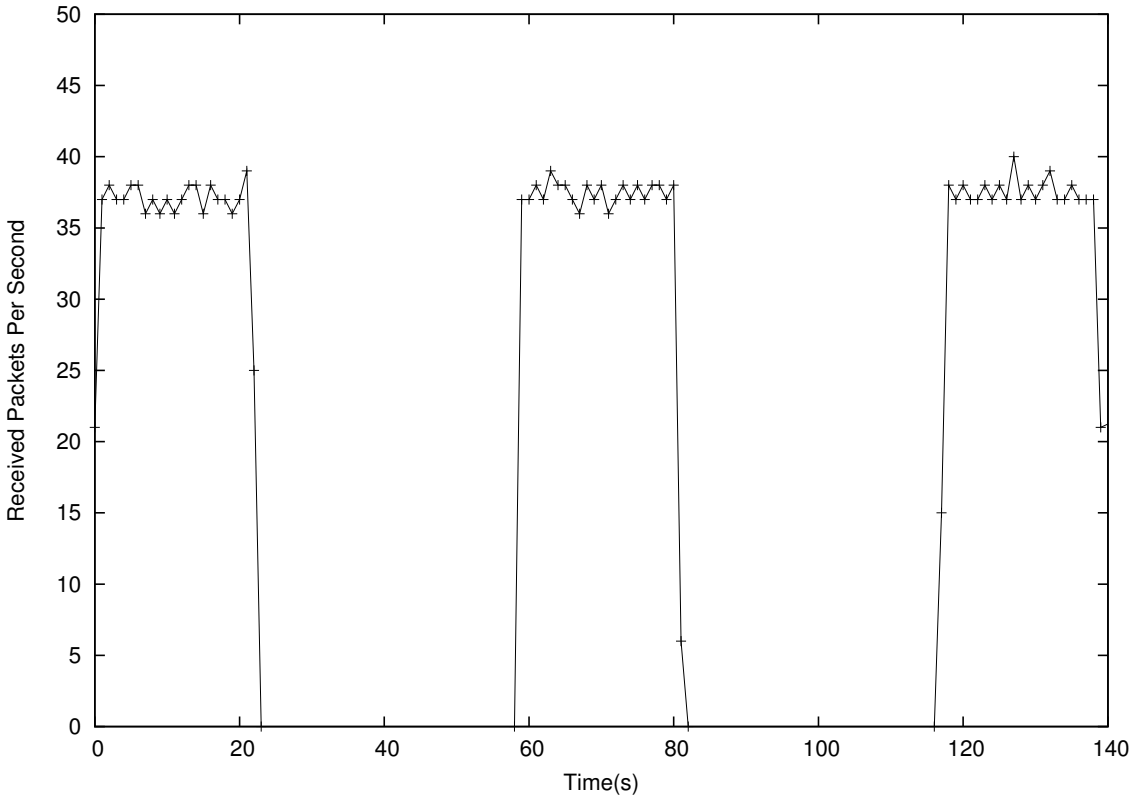
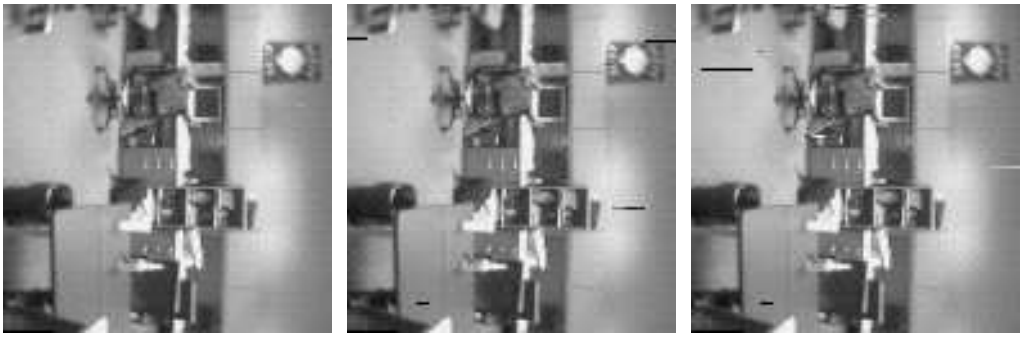


Figure 4.28: Packets/Second for the first 140 seconds of a multi-hop experimental run, with the interval between images set to 60 seconds.

Since each of our multi-hop data packets contain 20 bytes of payload data, as shown in Figure 4.24, a minimum of 820 packets are required to send the payload data. In the experiment depicted in Figure 4.28, the nodes sending the image through the network each sent between 825 and 840

packets per image.



(a) Image received without error

(b) Image received with packet errors

(c) Image received with missing packet

Figure 4.29: Sample images from a multi-hop test.

Figure 4.29 shows that the image is delivered over multiple hops,but that there is some loss of data. Most of this data loss is not due to missing packets, but due to corruption within a single packet. Figure 4.29(b) shows an image with sub-packet level errors. Occasionally, the error detecting code used to detect errors is insufficient for determining if transmission errors occur.

We also note that there are occasions when packets do not arrive at the destination by the time the image is created, there are two factors that lead to this problem. First, the frame program writes the image as soon as it receives the last packet. If a packet prior to the last arrives out of order, it is ignored. Second, the 5 retransmissions used by the *ReliableRoute* module is not sufficient for all cases, as shown in Figure 4.29(c).

Based on data gathered from these experiments and Equation 4.4 we have calculated the expected lifetime of an Image/Video sensor equipped node and the expected lifetime of a relay only node at radio power levels 11 and 31 with a 60 seconds between the start of each image. Since each node must send and receive data, we made the calculations assuming that half of the time

spent sending the image was used for transmission and half was used for receipt of data. This is a reasonable assumption, given that each node is capable of overhearing at least the next and previous hop neighbor nodes. These results appear in Table 4.8.

Table 4.8: Multi-Hop Power calculations based on Equation 4.4

Power Level	Wait Time (s)	Camera (Yes/No)	$C_{transmit}$ (mA)	$T_{transmit}$ (s)	$T_{receive}$ (s)	$T_{capture}$ (s)	T_{cycle} (s)	C_{total} (mA)	$\frac{1500mAh}{C_{total}}$ (h)
31	60	Yes	17.4	11	11	0.01	60	22.94	65.38
11	60	Yes	11.2	11	11	0.01	60	21.35	70.27
31	60	no	17.4	11	11	0	60	15.07	99.51
11	60	no	11.2	11	11	0	60	13.47	111.29

Due to the inability to provide a clean experimental space, as discussed in section 4.7.1, we have chosen not to experimentally determine the lifetime of the senor nodes in the multihop case. This is a subject deserving of further study, should appropriate facilities become available.

Discussion

5.1 Summary and Conclusions

We have shown that power consumption can be reduced when the node density increases in a wireless sensor network, provided the network transceivers can vary the power at which they operate. This decrease in power consumption implies a longer lifespan for the network nodes deployed in the field.

Furthermore, we have shown analytically the network capacity can increase when additional nodes are added to the network to increase the deployed density of the network. This increase in capacity is due directly to a reduction in interference between concurrent transmissions when the transceivers are operating at reduced power.

We have also discussed factors which limit the deployment density that can be applied based on external factors. From a technical perspective, the primary concern is putting an upper bound on the transmission latency based on time constrains of the application in question.

We have implemented a network testbed for further exploration of the topic of energy consumption in an Image/Video Wireless Sensor Network. This network allows reliably sending im-

ages through multiple hops to a sink node which can be accessed through the Internet, forming a Sensor Web. Utilizing this network, we have demonstrated that there is a trend towards power savings when the density of the network is increased.

Since we are able to maintain connectivity for an increased period of time using a high density deployment model, while at the same time increasing the effective capacity of the network as a whole, a high density sensor network should be suitable for high data volume sensors and sensor networks.

5.2 Contributions

We have made several contributions to the field of Image/Video Wireless Sensor Networks.

We have established a set of design criteria which must be considered for Image/Video Wireless Sensor Networks. This set of criteria includes: energy efficiency,capacity, sensor connectivity, sensor coverage, and latency.

We have developed a high density network deployment which we have shown, both analytically and experimentally, is capable of increasing energy efficiency in an Image Video Wireless Sensor Network. This network is capable of maintaining the capacity required to deliver data through the use of relay only nodes. Connectivity and coverage requirements provide a lower bound on the deployment density and latency requirements provide an upper bound.

We have developed a testbed for experimenting with Image/Video Wireless Sensor Networks. This testbed allows transmission of images through a network of a specified density to a sink node connected to the Internet. A collection of these networks form a Sensor Web, which may be

monitored from a centralized location.

To enable our Sensor Web, we have provided software, as seen in the Apendix, capable of capturing image data from a Cyclops camera and transmitting that image through a network of MICAz motes to a sink node. The software on the sink node is capable of reconstructing the image received through this wireless network.

Bibliography

[1] Tian He, Sudha Krishnamurthy, Liqian Luo, Ting Yan, Lin Gu, Radu Stoleru, Gang Zhou, Qing Cao, Pascal Vicaire, John A. Stankovic, Tarek F. Abdelzaher, Jonathan Hui, and Bruce Krogh. Vigilnet: An integrated sensor network system for energy-efficient surveillance. *ACM Trans. Sen. Netw.*, 2(1):1–38, 2006.

[2] Georga tech testbed for wireless multimedia sensor networks. http://www.ece.gatech.edu/research/labs/bwn/WMSN/testbed.html.

[3] Jinyang Li, Charles Blake, Douglas S.J. De Couto, Hu Imm Lee, and Robert Morris. Capacity of ad hoc wireless networks. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 61–69, New York, NY, USA, 2001. ACM Press.

[4] Crossbow micaz mote data sheet. http://www.xbow.com.

[5] Texas instraments/chipcon cc2420 data sheet. http://www.ti.com.

[6] Deernet. http://www.wu.ece.ufl.edu/projects/DeerNet/DeerNet.html.

[7] Cyclops camera. http://www.cyclopscamera.com/.

[8] Zoë; Abrams, Ashish Goel, and Serge Plotkin. Set k-cover algorithms for energy efficient monitoring in wireless sensor networks. In *IPSN '04: Proceedings of the third international symposium on Information processing in sensor networks*, pages 424–432, New York, NY, USA, 2004. ACM Press.

[9] Himanshu Gupta, Vishnu Navda, Samir R. Das, and Vishal Chowdhary. Efficient gathering of correlated data in sensor networks. In *MobiHoc '05: Proceedings of the 6th ACM international symposium on Mobile ad hoc networking and computing*, pages 402–413, New York, NY, USA, 2005. ACM Press.

[10] Sriram V. Pemmaraju and Imran A. Pirwani. Energy conservation via domatic partitions. In *MobiHoc '06: Proceedings of the seventh ACM international symposium on Mobile ad hoc networking and computing*, pages 143–154, New York, NY, USA, 2006. ACM Press.

[11] Curt Schurgers, Vlasios Tsiatsis, Saurabh Ganeriwal, and Mani Srivastava. Optimizing sensor networks in the energy-latency-density design space. *IEEE Transactions on Mobile Computing*, 1(1):70–80, 2002.

[12] Y. Xu, J. Heidemann, and D.Estrin. Adaptive energy-conserving routing for multihop ad hoc networks. Research Report 527, USC/ISI, October 2000.

[13] Ya Xu, John Heidemann, and Deborah Estrin. Geography-informed energy conservation for ad hoc routing. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 70–84, New York, NY, USA, 2001. ACM Press.

[14] A. Abbas, J. M. Bahi, and A. Mostefaoui. Optimizing energy consumption in wireless ad hoc networks. In *PE-WASUN '05: Proceedings of the 2nd ACM international workshop on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks*, pages 279–280, New York, NY, USA, 2005. ACM Press.

[15] Ali Iranli, Morteza Maleki, and Massoud Pedram. Energy efficient strategies for deployment of a two-level wireless sensor network. In *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pages 233–238, New York, NY, USA, 2005. ACM Press.

[16] Ram Ramanathan and Regina Hain. Topology control of multihop wireless networks using transmit power adjustment. In *INFOCOM (2)*, pages 404–413, 2000.

[17] Guoliang Xing, Chenyang Lu, Ying Zhang, Qingfeng Huang, and Robert Pless. Minimum power configuration for wireless communication in sensor networks. *ACM Trans. Sen. Netw.*, 3(2):11, 2007.

[18] Qunfeng Dong. Maximizing system lifetime in wireless sensor networks. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 3, Piscataway, NJ, USA, 2005. IEEE Press.

[19] P. Gupta and P. R. Kumar. The capacity of wireless networks. *IEEE Transactions on Information Theory*, 46(2):388–404, 2000.

[20] Benjie Chen, Kyle Jamieson, Hari Balakrishnan, and Robert Morris. Span: an energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. *Wirel. Netw.*, 8(5):481–494, 2002.

[21] Benjie Chen, Kyle Jamieson, Hari Balakrishnan, and Robert Morris. Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 85–96, New York, NY, USA, 2001. ACM Press.

[22] Nsf special report: The secret lives of wild anaimals. http://www.nsf.gov/news/special_reports/animals/index.jsp.

[23] Crossbow stargate gateway data sheet. http://www.xbow.com.

[24] Shan Lin, Jingbin Zhang, Gang Zhou, Lin Gu, John A. Stankovic, and Tian He. Atpc: adaptive transmission power control for wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 223–236, New York, NY, USA, 2006. ACM Press.

[25] Luiz H. A. Correia, Daniel F. Macedo, Daniel A. C. Silva, Aldri L. dos Santos, Antonio A. F. Loureiro, and José Marcos S. Nogueira. Transmission power control in mac protocols for wireless sensor networks. In *MSWiM '05: Proceedings of the 8th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pages 282–289, New York, NY, USA, 2005. ACM.

[26] Ramin Hekmat and Piet Van Mieghem. Interference in wireless multi-hop ad-hoc networks and its effect on network capacity. *Wirel. Netw.*, 10(4):389–399, 2004.

[27] Theodore S. Rappaport. *Wireless Communications, Principles and Practice*. Pretince Hall PTR, Upper Saddle River,NJ, second edition, 2002.

[28] Crossbow mib510 serial gateway data sheet. http://www.xbow.com.

Appendix A:Single Hop Reception Program - Host Computer

A.1 frame.c

```
//$Header: /home/cvs/cvsroot/tos-contrib/cyclops/tools/R/frame.c,
// v 1.2 2005/04/19 23:30:44 local Exp $
//Modified by: Mohammad Rahimi mhr@cens.ucla.edu
//Modified by Paul Bender bender.13@wright.edu to work with MICAz

#include "serial.h"
#include "serialDump.h"

typedef struct HEADER{
    unsigned short int type;                /* Magic identifier */
    unsigned int size;                      /*File size in bytes*/
    unsigned short int reserved1, reserved2;
    unsigned int offset;                    /* Offset to image
                                               data, bytes */
}  __attribute__((packed)) HEADER;

typedef struct INFOHEADER {
    unsigned int size;              /* Header size in bytes    */
    int width,height;               /*Width and height of image*/
    unsigned short int planes;      /* Number of colour planes */
    unsigned short int bits;        /* Bits per pixel          */
    unsigned int compression;       /* Compression type        */
    unsigned int imagesize;         /* Image size in bytes     */
    int xresolution,yresolution;    /* Pixels per meter        */
    unsigned int ncolours;          /* Number of colours       */
    unsigned int importantcolours;  /* Important colours       */
} INFOHEADER;

//**************************************************************
//This function grabs a packet
```

```
//*************************************************************
TOS_SERIAL_Msg read_packet()
{
int count;
char c;
        char input_buffer[sizeof(TOS_SERIAL_Msg)];
        TOS_SERIAL_Msg msg;
bzero(input_buffer,sizeof(TOS_SERIAL_Msg));

//search through to find 0x7e signifing the start of a
        //packet
    while(input_buffer[0] != (char)(0x7e)){
        while((c = read(input_stream, input_buffer, 1)) != 1){
        };
    }
    count = 1;
    //you have the first byte now read the rest.
    while(count <sizeof(TOS_SERIAL_Msg) ) {
        count += read(input_stream, input_buffer + count,1);
    }
    memcpy(&msg,input_buffer,sizeof(TOS_SERIAL_Msg));
    return msg;
}

//*************************************************************
//This function prints a packet
//*************************************************************
void print_packet(TOS_SERIAL_Msg myMsg)
{
    int i;
    char input_buffer[sizeof(TOS_SERIAL_Msg)];
    memcpy(input_buffer,&myMsg,sizeof(TOS_SERIAL_Msg));
    for(i=0; i <sizeof(TOS_SERIAL_Msg); i++)
           printf("%02x ", (unsigned char) input_buffer[i]);
    printf("\n");fflush(stdout);
    return;
}

void print_data(TOS_SERIAL_Msg myMsg)
{
    int i;
    char input_buffer[29];
    memcpy(input_buffer,&myMsg.data,29);
    for(i=0; i <29; i++)
        printf("%02x ", (unsigned char) input_buffer[i]);
```

```
    printf("\n");fflush(stdout);
    return;
}
//*************************************************************
//This function prints a packet
//*************************************************************
void print_dump(char *myBuf,int myBufSize)
{
  int i;
  printf("\n\n*****************serial dump*****************\n");
    for(i=0;i<myBufSize; i++)
        {
            if(i%16==0) printf("\n");
            printf("%2x ", (unsigned char) myBuf[i]);
        }
    printf("\n*******************end*******************\n");
    fflush(stdout);
    return;
}

//*************************************************************
//This function prints a packet
//*************************************************************
void print_file_dump(char *myBuf,int myBufSize)
{
    int i;
    FILE *fp1,*fp2;
    if((fp1 = fopen ("frame.dat", "w"))==NULL)
        printf("Data file error\n");
//printf("\n\n*****************serial dump*****************\n");
    for(i=0;i<myBufSize; i++)
        {
            if(i%16==0 && i!=0) fprintf(fp1,"\n");
            fprintf(fp1,"%2x ", (unsigned char) myBuf[i]);
        }
    //printf("\n*******************end*******************\n");
    //fflush(stdout);
    fclose(fp1);
    //we only create lock file
    fp2 = fopen (".frame.Lock", "w");
    fclose(fp2);
    return;
}

//*************************************************************
```

```
//This function creates a bmp file of the image
//*************************************************************
void print_bmp_dump(char *myBuf, int myBufSize){
    FILE *fp;
    HEADER h1;
    INFOHEADER h2;
    //char *outputBuf;
    unsigned char palette[] = { \
                                0x00, 0x00, 0x00, 0x00, 0x01, 0x01,\
                                0x01, 0x00, 0x02, 0x02, 0x02, 0x00,\
                                0x03, 0x03, 0x03, 0x00, 0x04, 0x04,\
                                0x04, 0x00, 0x05, 0x05, 0x05, 0x00,\
                                0x06, 0x06, 0x06, 0x00, 0x07, 0x07,\
                                0x07, 0x00, 0x08, 0x08, 0x08, 0x00,\
                                0x09, 0x09, 0x09, 0x00, 0x0a, 0x0a,\
                                0x0a, 0x00, 0x0b, 0x0b, 0x0b, 0x00,\
                                0x0c, 0x0c, 0x0c, 0x00, 0x0d, 0x0d,\
                                0x0d, 0x00, 0x0e, 0x0e, 0x0e, 0x00,\
                                0x0f, 0x0f, 0x0f, 0x00, 0x10, 0x10,\
                                0x10, 0x00, 0x11, 0x11, 0x11, 0x00,\
                                0x12, 0x12, 0x12, 0x00, 0x13, 0x13,\
                                0x13, 0x00, 0x14, 0x14, 0x14, 0x00,\
                                0x15, 0x15, 0x15, 0x00, 0x16, 0x16,\
                                0x16, 0x00, 0x17, 0x17, 0x17, 0x00,\
                                0x18, 0x18, 0x18, 0x00, 0x19, 0x19,\
                                0x19, 0x00, 0x1a, 0x1a, 0x1a, 0x00,\
                                0x1b, 0x1b, 0x1b, 0x00, 0x1c, 0x1c,\
                                0x1c, 0x00, 0x1d, 0x1d, 0x1d, 0x00,\
                                0x1e, 0x1e, 0x1e, 0x00, 0x1f, 0x1f,\
                                0x1f, 0x00, 0x20, 0x20, 0x20, 0x00,\
                                0x21, 0x21, 0x21, 0x00, 0x22, 0x22,\
                                0x22, 0x00, 0x23, 0x23, 0x23, 0x00,\
                                0x24, 0x24, 0x24, 0x00, 0x25, 0x25,\
                                0x25, 0x00, 0x26, 0x26, 0x26, 0x00,\
                                0x27, 0x27, 0x27, 0x00, 0x28, 0x28,\
                                0x28, 0x00, 0x29, 0x29, 0x29, 0x00,\
                                0x2a, 0x2a, 0x2a, 0x00, 0x2b, 0x2b,\
                                0x2b, 0x00, 0x2c, 0x2c, 0x2c, 0x00,\
                                0x2d, 0x2d, 0x2d, 0x00, 0x2e, 0x2e,\
                                0x2e, 0x00, 0x2f, 0x2f, 0x2f, 0x00,\
                                0x30, 0x30, 0x30, 0x00, 0x31, 0x31,\
                                0x31, 0x00, 0x32, 0x32, 0x32, 0x00,\
                                0x33, 0x33, 0x33, 0x00, 0x34, 0x34,\
                                0x34, 0x00, 0x35, 0x35, 0x35, 0x00,\
                                0x36, 0x36, 0x36, 0x00, 0x37, 0x37,\
```

```
0x37, 0x00, 0x38, 0x38, 0x38, 0x00,\
0x39, 0x39, 0x39, 0x00, 0x3a, 0x3a,\
0x3a, 0x00, 0x3b, 0x3b, 0x3b, 0x00,\
0x3c, 0x3c, 0x3c, 0x00, 0x3d, 0x3d,\
0x3d, 0x00, 0x3e, 0x3e, 0x3e, 0x00,\
0x3f, 0x3f, 0x3f, 0x00, 0x40, 0x40,\
0x40, 0x00, 0x41, 0x41, 0x41, 0x00,\
0x42, 0x42, 0x42, 0x00, 0x43, 0x43,\
0x43, 0x00, 0x44, 0x44, 0x44, 0x00,\
0x45, 0x45, 0x45, 0x00, 0x46, 0x46,\
0x46, 0x00, 0x47, 0x47, 0x47, 0x00,\
0x48, 0x48, 0x48, 0x00, 0x49, 0x49,\
0x49, 0x00, 0x4a, 0x4a, 0x4a, 0x00,\
0x4b, 0x4b, 0x4b, 0x00, 0x4c, 0x4c,\
0x4c, 0x00, 0x4d, 0x4d, 0x4d, 0x00,\
0x4e, 0x4e, 0x4e, 0x00, 0x4f, 0x4f,\
0x4f, 0x00, 0x50, 0x50, 0x50, 0x00,\
0x51, 0x51, 0x51, 0x00, 0x52, 0x52,\
0x52, 0x00, 0x53, 0x53, 0x53, 0x00,\
0x54, 0x54, 0x54, 0x00, 0x55, 0x55,\
0x55, 0x00, 0x56, 0x56, 0x56, 0x00,\
0x57, 0x57, 0x57, 0x00, 0x58, 0x58,\
0x58, 0x00, 0x59, 0x59, 0x59, 0x00,\
0x5a, 0x5a, 0x5a, 0x00, 0x5b, 0x5b,\
0x5b, 0x00, 0x5c, 0x5c, 0x5c, 0x00,\
0x5d, 0x5d, 0x5d, 0x00, 0x5e, 0x5e,\
0x5e, 0x00, 0x5f, 0x5f, 0x5f, 0x00,\
0x60, 0x60, 0x60, 0x00, 0x61, 0x61,\
0x61, 0x00, 0x62, 0x62, 0x62, 0x00,\
0x63, 0x63, 0x63, 0x00, 0x64, 0x64,\
0x64, 0x00, 0x65, 0x65, 0x65, 0x00,\
0x66, 0x66, 0x66, 0x00, 0x67, 0x67,\
0x67, 0x00, 0x68, 0x68, 0x68, 0x00,\
0x69, 0x69, 0x69, 0x00, 0x6a, 0x6a,\
0x6a, 0x00, 0x6b, 0x6b, 0x6b, 0x00,\
0x6c, 0x6c, 0x6c, 0x00, 0x6d, 0x6d,\
0x6d, 0x00, 0x6e, 0x6e, 0x6e, 0x00,\
0x6f, 0x6f, 0x6f, 0x00, 0x70, 0x70,\
0x70, 0x00, 0x71, 0x71, 0x71, 0x00,\
0x72, 0x72, 0x72, 0x00, 0x73, 0x73,\
0x73, 0x00, 0x74, 0x74, 0x74, 0x00,\
0x75, 0x75, 0x75, 0x00, 0x76, 0x76,\
0x76, 0x00, 0x77, 0x77, 0x77, 0x00,\
0x78, 0x78, 0x78, 0x00, 0x79, 0x79,\
0x79, 0x00, 0x7a, 0x7a, 0x7a, 0x00,\
```

```
0x7b, 0x7b, 0x7b, 0x00, 0x7c, 0x7c,\
0x7c, 0x00, 0x7d, 0x7d, 0x7d, 0x00,\
0x7e, 0x7e, 0x7e, 0x00, 0x7f, 0x7f,\
0x7f, 0x00, 0x80, 0x80, 0x80, 0x00,\
0x81, 0x81, 0x81, 0x00, 0x82, 0x82,\
0x82, 0x00, 0x83, 0x83, 0x83, 0x00,\
0x84, 0x84, 0x84, 0x00, 0x85, 0x85,\
0x85, 0x00, 0x86, 0x86, 0x86, 0x00,\
0x87, 0x87, 0x87, 0x00, 0x88, 0x88,\
0x88, 0x00, 0x89, 0x89, 0x89, 0x00,\
0x8a, 0x8a, 0x8a, 0x00, 0x8b, 0x8b,\
0x8b, 0x00, 0x8c, 0x8c, 0x8c, 0x00,\
0x8d, 0x8d, 0x8d, 0x00, 0x8e, 0x8e,\
0x8e, 0x00, 0x8f, 0x8f, 0x8f, 0x00,\
0x90, 0x90, 0x90, 0x00, 0x91, 0x91,\
0x91, 0x00, 0x92, 0x92, 0x92, 0x00,\
0x93, 0x93, 0x93, 0x00, 0x94, 0x94,\
0x94, 0x00, 0x95, 0x95, 0x95, 0x00,\
0x96, 0x96, 0x96, 0x00, 0x97, 0x97,\
0x97, 0x00, 0x98, 0x98, 0x98, 0x00,\
0x99, 0x99, 0x99, 0x00, 0x9a, 0x9a,\
0x9a, 0x00, 0x9b, 0x9b, 0x9b, 0x00,\
0x9c, 0x9c, 0x9c, 0x00, 0x9d, 0x9d,\
0x9d, 0x00, 0x9e, 0x9e, 0x9e, 0x00,\
0x9f, 0x9f, 0x9f, 0x00, 0xa0, 0xa0,\
0xa0, 0x00, 0xa1, 0xa1, 0xa1, 0x00,\
0xa2, 0xa2, 0xa2, 0x00, 0xa3, 0xa3,\
0xa3, 0x00, 0xa4, 0xa4, 0xa4, 0x00,\
0xa5, 0xa5, 0xa5, 0x00, 0xa6, 0xa6,\
0xa6, 0x00, 0xa7, 0xa7, 0xa7, 0x00,\
0xa8, 0xa8, 0xa8, 0x00, 0xa9, 0xa9,\
0xa9, 0x00, 0xaa, 0xaa, 0xaa, 0x00,\
0xab, 0xab, 0xab, 0x00, 0xac, 0xac,\
0xac, 0x00, 0xad, 0xad, 0xad, 0x00,\
0xae, 0xae, 0xae, 0x00, 0xaf, 0xaf,\
0xaf, 0x00, 0xb0, 0xb0, 0xb0, 0x00,\
0xb1, 0xb1, 0xb1, 0x00, 0xb2, 0xb2,\
0xb2, 0x00, 0xb3, 0xb3, 0xb3, 0x00,\
0xb4, 0xb4, 0xb4, 0x00, 0xb5, 0xb5,\
0xb5, 0x00, 0xb6, 0xb6, 0xb6, 0x00,\
0xb7, 0xb7, 0xb7, 0x00, 0xb8, 0xb8,\
0xb8, 0x00, 0xb9, 0xb9, 0xb9, 0x00,\
0xba, 0xba, 0xba, 0x00, 0xbb, 0xbb,\
0xbb, 0x00, 0xbc, 0xbc, 0xbc, 0x00,\
0xbd, 0xbd, 0xbd, 0x00, 0xbe, 0xbe,\
```

```
                                0xbe, 0x00, 0xbf, 0xbf, 0xbf, 0x00,\
                                0xc0, 0xc0, 0xc0, 0x00, 0xc1, 0xc1,\
                                0xc1, 0x00, 0xc2, 0xc2, 0xc2, 0x00,\
                                0xc3, 0xc3, 0xc3, 0x00, 0xc4, 0xc4,\
                                0xc4, 0x00, 0xc5, 0xc5, 0xc5, 0x00,\
                                0xc6, 0xc6, 0xc6, 0x00, 0xc7, 0xc7,\
                                0xc7, 0x00, 0xc8, 0xc8, 0xc8, 0x00,\
                                0xc9, 0xc9, 0xc9, 0x00, 0xca, 0xca,\
                                0xca, 0x00, 0xcb, 0xcb, 0xcb, 0x00,\
                                0xcc, 0xcc, 0xcc, 0x00, 0xcd, 0xcd,\
                                0xcd, 0x00, 0xce, 0xce, 0xce, 0x00,\
                                0xcf, 0xcf, 0xcf, 0x00, 0xd0, 0xd0,\
                                0xd0, 0x00, 0xd1, 0xd1, 0xd1, 0x00,\
                                0xd2, 0xd2, 0xd2, 0x00, 0xd3, 0xd3,\
                                0xd3, 0x00, 0xd4, 0xd4, 0xd4, 0x00,\
                                0xd5, 0xd5, 0xd5, 0x00, 0xd6, 0xd6,\
                                0xd6, 0x00, 0xd7, 0xd7, 0xd7, 0x00,\
                                0xd8, 0xd8, 0xd8, 0x00, 0xd9, 0xd9,\
                                0xd9, 0x00, 0xda, 0xda, 0xda, 0x00,\
                                0xdb, 0xdb, 0xdb, 0x00, 0xdc, 0xdc,\
                                0xdc, 0x00, 0xdd, 0xdd, 0xdd, 0x00,\
                                0xde, 0xde, 0xde, 0x00, 0xdf, 0xdf,\
                                0xdf, 0x00, 0xe0, 0xe0, 0xe0, 0x00,\
                                0xe1, 0xe1, 0xe1, 0x00, 0xe2, 0xe2,\
                                0xe2, 0x00, 0xe3, 0xe3, 0xe3, 0x00,\
                                0xe4, 0xe4, 0xe4, 0x00, 0xe5, 0xe5,\
                                0xe5, 0x00, 0xe6, 0xe6, 0xe6, 0x00,\
                                0xe7, 0xe7, 0xe7, 0x00, 0xe8, 0xe8,\
                                0xe8, 0x00, 0xe9, 0xe9, 0xe9, 0x00,\
                                0xea, 0xea, 0xea, 0x00, 0xeb, 0xeb,\
                                0xeb, 0x00, 0xec, 0xec, 0xec, 0x00,\
                                0xed, 0xed, 0xed, 0x00, 0xee, 0xee,\
                                0xee, 0x00, 0xef, 0xef, 0xef, 0x00,\
                                0xf0, 0xf0, 0xf0, 0x00, 0xf1, 0xf1,\
                                0xf1, 0x00, 0xf2, 0xf2, 0xf2, 0x00,\
                                0xf3, 0xf3, 0xf3, 0x00, 0xf4, 0xf4,\
                                0xf4, 0x00, 0xf5, 0xf5, 0xf5, 0x00,\
                                0xf6, 0xf6, 0xf6, 0x00, 0xf7, 0xf7,\
                                0xf7, 0x00, 0xf8, 0xf8, 0xf8, 0x00,\
                                0xf9, 0xf9, 0xf9, 0x00, 0xfa, 0xfa,\
                                0xfa, 0x00, 0xfb, 0xfb, 0xfb, 0x00,\
                                0xfc, 0xfc, 0xfc, 0x00, 0xfd, 0xfd,\
                                0xfd, 0x00, 0xfe, 0xfe, 0xfe, 0x00,\
                                0xff, 0xff, 0xff, 0x00,
};
```

```
int i,i2;
char tmp;
char filename[60]="\0";
char *newname;
char template[20]="frameXXXXXX";
//BW or Color image
int format;
//Demensions of image
int width;
int height;
int align = 0;  // needed for both types of images
int padding =0; // needed by color images

//We pick based on image size. We assume that images can
//either be 3 byte RGB or 1 byte Black and White

//Black and white images
if((myBufSize == 1024)||(myBufSize == 4096)||
   (myBufSize == 16384)) {
    format = 0;
}
//RGB color images
else if ((myBufSize == 3072)||(myBufSize ==12288)||
         (myBufSize == 49152))  {
    format = 1;
}
//We cannot create an image, the buffer size is not correct
else {
    return;
}

switch(myBufSize) {
case 1024:
case 3072:
    width = 32;
    height = 32;
    break;
case 4096:
case 12288:
    width = 64;
    height = 64;
    break;
case 16384:
case 49152:
```

```
        width = 128;
        height = 128;
    }

    // Open a file to save data
    //newname=mktemp(template);
    //strcat(filename,newname);
    //strcat(filename,".bmp");
    sprintf(filename,"frame-%i.bmp",time(NULL));
    if((fp = fopen (filename, "wb"))==NULL)
        {
            printf("Output Data file error\n");
            exit(1);
        }

    /********Black/White image*********/
    if (format == 0){
        align = ((height*width)+54) % 4;

        h1.type = 19778; //'M' * 256 + 'B';
        h1.size = (width*height) + 54 + align;
        h1.reserved1 = 0;
        h1.reserved2 = 0;
        h1.offset = 54+256*4;

        h2.size = 40;
        h2.width = width;
        h2.height = height;
        h2.planes = 1;
        h2.bits = 8;
        h2.compression = 0;
        h2.imagesize = 0;
        h2.xresolution = 0;
        h2.yresolution = 0;
        h2.ncolours = 256;
        h2.importantcolours = 0;
    }

    /************COLOR image**********/
    if (format == 1){
        h1.type = 19778; //'M' * 256 + 'B';
        h1.size = (width*height*3) + 54;
        h1.reserved1 = 0;
        h1.reserved2 = 0;
        h1.offset = 54;
```

```
        h2.size = 40;
        h2.width = width;
        h2.height = height;
        h2.planes = 1;
        h2.bits = 24;
        h2.compression = 0;
        h2.imagesize = 0;
        h2.xresolution = 0;
        h2.yresolution = 0;
        h2.ncolours = 0;
        h2.importantcolours = 0;

        i2=1;
        //This is necessary to get the RGB values in the right
        //  order for BMP.  Apparently cyclops does not provide
        //  the appropriate ordering of the RGB values for BMP
        //  conversion
        for(i=0;i<width*height*3; i++)
     {
          if (i2 % 3 == 0)
         {
           tmp = myBuf[i];
           myBuf[i] = myBuf[i-2];
           myBuf[i-2]=tmp;
           }
          i2++;
     }
    }

    fwrite (&h1, sizeof(HEADER), 1, fp);
    fwrite (&h2, sizeof(INFOHEADER), 1, fp);

    if (format == 0)  //Insert Color Pallete only if BW image
        {
            fwrite (palette, sizeof(char), 256*4, fp);
            fwrite (myBuf, sizeof(char), width*height, fp);
        }

    if (format == 1)
        fwrite (myBuf, sizeof(char), width*height*3, fp);

    fclose(fp);

}
```

```
//*************************************************************
//This function prints progress
//*************************************************************
void print_progress(TOS_SERIAL_Msg myMsg)
{
    printf(".");fflush(stdout);
    return;
}


//*************************************************************
//This function returns amount of necessary data for the dump
//*************************************************************
int getSize(TOS_SERIAL_Msg myMsg)
{
    serialDumpHeader_t myHeader;
    memcpy(&myHeader,&myMsg.data,sizeof(serialDumpHeader_t));
    printf("\nreceiving buffer of size:%d ->",myHeader.seq);
    fflush(stdout);
    //printf("received buffer size:%02x:%02x\n",
            (char)(myHeader.seq),(char)(myHeader.seq>>8));
    return myHeader.seq;
}

//*************************************************************
//This function returns amount of necessary data for the dump
//*************************************************************
mode append(char *buf,TOS_SERIAL_Msg myMsg,int size)
{
    serialDumpHeader_t myHeader;
    int i;
    memcpy(&myHeader,&myMsg.data,sizeof(serialDumpHeader_t));
    //printf("\nheadersize %d, Msg size %d, data size %d,"
    //        sizeof(serialDumpHeader_t),sizeof(myMsg),
    //        sizeof(myMsg.data));
    //printf("sequence no:%d (%02x %02x)\n",myHeader.seq,
    //        (myHeader.seq&0xff00)>>8,(myHeader.seq&0x00ff));

    if(myHeader.seq>size) return PACKET_ERROR;

    if( myHeader.seq >
            (TOSH_DATA_LENGTH-sizeof(serialDumpHeader_t)) )
          //this is not the last packet
        {
```

```
            memcpy(buf+(size-myHeader.seq),
                   myMsg.data+sizeof(serialDumpHeader_t),
                   (TOSH_DATA_LENGTH-sizeof(serialDumpHeader_t)));
            return GET_NEXT_PACKET;
        }
    else  //this is the last packet
        {
            memcpy(buf+(size-myHeader.seq),
                  myMsg.data+sizeof(serialDumpHeader_t),
                  myHeader.seq);
            return IDLE;
        }
}

//*****************************************************************
//Main
//*****************************************************************

int main(int argc, char ** argv)
{
    FILE *tsfp;
    char tsfilename[100];
    int n = 0;
    unsigned long packetcount = 0;
    unsigned long oldcount = 0;
    unsigned char *buf;
    int buffSize;
    mode myMode=IDLE;
    bool verberose= FALSE;

    if (argc > 4 || argc > 1 && argv[1][0] == '-') {
        print_usage(argv[0]);
        exit(2);
    }
    if (argc==4 && argv[3][1]=='t') {
        verberose=TRUE;
    } else if(argc==4) {
        print_usage(argv[0]);
        exit(2);
    }
    open_input(argc, argv);

    sprintf(tsfilename,"timestamp-%u.dat",time(NULL));
    if((tsfp = fopen (tsfilename,"wb"))==NULL)
        {
```

```
            printf("Output Data file error\n");
            exit(1);
        }

    //Here we run the state machine forever
    while(1) {
        TOS_SERIAL_Msg msg;
        msg = read_packet();
        if(myMode==IDLE)
            {
                buffSize=getSize(msg);
                if(buffSize>0)
                    {
                        myMode=GET_NEXT_PACKET;
                        buf = (char *) malloc(buffSize);
                        bzero(buf,buffSize);
                    }
            }
        if(verberose) print_packet(msg);else print_progress(msg);
        //if(verberose) print_data(msg);
        switch(append(buf,msg,buffSize))
            {
            case GET_NEXT_PACKET:
                packetcount++;
                if(packetcount<oldcount+100000){
                        fprintf(tsfp,"%u %i\n",time(NULL),
                               (int8_t)msg.rssi-45);
                 fflush(tsfp);
                } else {

                       fclose(tsfp);
                       sprintf(tsfilename,"timestamp-%u.dat",
                       time(NULL));
                       if((tsfp = fopen (tsfilename,"wb"))==NULL)
                {
                              printf("Output Data file error\n");
                              exit(1);
                       }
                       fprintf(tsfp,"%u %i\n",time(NULL),
                              (int8_t)msg.rssi-45);
                       fflush(tsfp);
                        oldcount=packetcount;
                }
                break;
            case IDLE:
```

```
            packetcount++;
            if(packetcount<oldcount+100000){
                    fprintf(tsfp,"%u %i\n",time(NULL),
                            (int8_t)msg.rssi-45);
             fflush(tsfp);
            } else {
                    fclose(tsfp);
                    sprintf(tsfilename,"timestamp-%u.dat",
                            time(NULL));
                    if((tsfp = fopen(tsfilename,"wb"))==NULL)
                    {
                        printf("Output Data file error\n");
                        exit(1);
                    }
                    fprintf(tsfp,"%u %i\n",time(NULL),
                            (int8_t)msg.rssi-45);
             fflush(tsfp);
                    oldcount=packetcount;
            }
            print_dump(buf,buffSize);
            print_file_dump(buf,buffSize);
            print_bmp_dump(buf,buffSize);
            printf("%u\n",packetcount);
            myMode=IDLE;
            free(buf);
            break;
        case PACKET_ERROR:
        default:
            packetcount++;
            if(packetcount<oldcount+100000){
                    fprintf(tsfp,"%u %i\n",time(NULL),
                            (int8_t)msg.rssi-45);
             fflush(tsfp);
            }
            else {
                    fclose(tsfp);
                    sprintf(tsfilename,"timestamp-%u.dat",
                            time(NULL));
                    if((tsfp = fopen(tsfilename,"wb"))==NULL)
                    {
                         printf("Output Data file error\n");
                         exit(1);
                    }
                    fprintf(tsfp,"%u %i\n",time(NULL),
                            (int8_t)msg.rssi-45);
```

```
                        fflush(tsfp);
                        oldcount=packetcount;
                }
                printf("!");
                //printf("an error encountered\n");
                //myMode=IDLE;
            }
    }
}
```

A.2 def.h

```
//Modified by Paul Bender bender.13@wright.edu to work with MICAz
#ifndef DEF_H
#define DEF_H

#define BAUDRATE B4800 //the default baudrate that the device is
                       //talking
#define SERIAL_DEVICE "/dev/ttyS0" //the port to use.

typedef unsigned char bool;
#define TRUE 1
#define FALSE 0

typedef enum { IDLE=12, GET_NEXT_PACKET , PACKET_ERROR} mode;
#define TOSH_DATA_LENGTH 29
#define uint8_t unsigned char
#define uint16_t unsigned short
//#include "AM.h"

//I hate to do that but Unfortunately there is no data structure
//I can grab from AM.h that has the serial port content. I am
//sorry so I had to create a data structure here. This means that
//if the actual data structure change this code will be broken.
//But this is very unlikely.
typedef struct TOS_SERIAL_Msg
{
  /*TOS_MSG data */
  uint16_t addr;
  uint8_t type;
  uint8_t group;
  //uint8_t length;
  /* Payload */
```

```
  int8_t data[TOSH_DATA_LENGTH];
  int8_t rssi; // RSSI value +45, subtract 45 to get the
               // real value - Paul
  uint16_t crc;
}TOS_SERIAL_Msg;

int input_stream;

void print_usage(char *name);
void open_input(int argc, char **argv);
TOS_SERIAL_Msg read_packet();
int getSize(TOS_SERIAL_Msg myMsg);
void print_progress(TOS_SERIAL_Msg myMsg);

#endif
```

A.3 serial.h

```
//$Header: /home/cvs/cvsroot/tos-contrib/cyclops/tools/R/serial.h
,v 1.1 2005/04/14 23:01:50 local Exp $
//Modified by: Mohammad Rahimi mhr@cens.ucla.edu

#ifndef SERIAL_H
#define SERIAL_H

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#ifdef __CYGWIN__
#include <windows.h>
#endif
#include "def.h"


void print_usage(char *name){
    //usage...
    fprintf(stderr,"usage: %s [serial port] [baudrate]\n", name);
    fprintf(stderr,"Default serial port is " SERIAL_DEVICE \
                   ", default baud rate is 19200\n");
```

```
}


void open_input(int argc, char **argv) {
    /* open input_stream for read/write */
    struct termios newtio;
    const char *name = SERIAL_DEVICE;
    unsigned long baudrate = BAUDRATE;

    if (argc > 1)
name = argv[1];
    if (argc > 2) {
int reqrate = atoi(argv[2]);

switch (reqrate) {
#ifdef B50
case 50: baudrate = B50; break;
#endif
#ifdef B75
case 75: baudrate = B75; break;
#endif
#ifdef B110
case 110: baudrate = B110; break;
#endif
#ifdef B134
case 134: baudrate = B134; break;
#endif
#ifdef B150
case 150: baudrate = B150; break;
#endif
#ifdef B200
case 200: baudrate = B200; break;
#endif
#ifdef B300
case 300: baudrate = B300; break;
#endif
#ifdef B600
case 600: baudrate = B600; break;
#endif
#ifdef B1200
case 1200: baudrate = B1200; break;
#endif
#ifdef B1800
case 1800: baudrate = B1800; break;
#endif
```

```
#ifdef B2400
case 2400: baudrate = B2400; break;
#endif
#ifdef B4800
case 4800: baudrate = B4800; break;
#endif// $Id: serial.h,v 1.1 2005/04/14 23:01:50 local Exp $
#ifdef B9600
case 9600: baudrate = B9600; break;
#endif
#ifdef B19200
case 19200: baudrate = B19200; break;
#endif
#ifdef B38400
case 38400: baudrate = B38400; break;
#endif
#ifdef B57600
case 57600: baudrate = B57600; break;
#endif
#ifdef B115200
case 115200: baudrate = B115200; break;
#endif
#ifdef B230400
case 230400: baudrate = B230400; break;
#endif
#ifdef B460800
case 460800: baudrate = B460800; break;
#endif
#ifdef B500000
case 500000: baudrate = B500000; break;
#endif
#ifdef B576000
case 576000: baudrate = B576000; break;
#endif
#ifdef B921600
case 921600: baudrate = B921600; break;
#endif
#ifdef B1000000
case 1000000: baudrate = B1000000; break;
#endif
#ifdef B1152000
case 1152000: baudrate = B1152000; break;
#endif
#ifdef B1500000
case 1500000: baudrate = B1500000; break;
#endif
```

```
#ifdef B2000000
case 2000000: baudrate = B2000000; break;
#endif
#ifdef B2500000
case 2500000: baudrate = B2500000; break;
#endif
#ifdef B3000000
case 3000000: baudrate = B3000000; break;
#endif
#ifdef B3500000
case 3500000: baudrate = B3500000; break;
#endif
#ifdef B4000000
case 4000000: baudrate = B4000000; break;
#endif
        default:
             fprintf(stderr,
                    "Unknown baudrate %s, defaulting to 19200\n",
                    argv[2]);
        }
    }

    input_stream = open(name, O_RDWR|O_NOCTTY);
    if (input_stream == -1) {
fprintf(stderr, "Failed to open %s", name);
perror("");
fprintf(stderr,
          "Make sure the user has permission to open device.\n");
exit(2);
    }
    printf("%s input_stream opened\n", name);
#ifdef __CYGWIN__
        /* For some very mysterious reason, this incantation is
           necessary to make the serial port work under some
           windows machines */
        HANDLE handle = (HANDLE)get_osfhandle(input_stream);
        DCB dcb;
        if (!(GetCommState(handle, &dcb) &&
              SetCommState(handle, &dcb))) {
                fprintf(stderr,
                        "serial port initialisation problem\n");
                exit(2);
         }
#endif
```

```
    /* Serial port setting */
    bzero(&newtio, sizeof(newtio));
    newtio.c_cflag = CS8 | CLOCAL | CREAD;
    newtio.c_iflag = IGNPAR | IGNBRK;
    cfsetispeed(&newtio, baudrate);
    cfsetospeed(&newtio, baudrate);

    tcflush(input_stream, TCIFLUSH);
    tcsetattr(input_stream, TCSANOW, &newtio);
}


#endif
```

Appendix B:Single Hop with Camera - unmodified

B.1 MoteI2CRelayC.nc

```
// Modified by Paul Bender (bender.13@wright.edu) to send an
// image using a MICAz mote from a Cyclops camera

includes MoteI2CRelay;

configuration MoteI2CRelayC { }
implementation {
    components
        Main,
        MoteI2CRelayM,
        GenericComm as Comm,
        LedsC,
        TimerC, I2CPacketMasterC;

    Main.StdControl -> MoteI2CRelayM;
    Main.StdControl -> TimerC;
    Main.StdControl -> Comm;


    MoteI2CRelayM.Leds -> LedsC;
    MoteI2CRelayM.Timer -> TimerC.Timer[unique("Timer")];

    //RF communication facility
    MoteI2CRelayM.CommControl -> Comm;
    MoteI2CRelayM.SendMsg -> Comm.SendMsg[Sample_Packet];
    MoteI2CRelayM.ReceiveMsg -> Comm.ReceiveMsg[Sample_Packet];

    //Communcation over I2C
    MoteI2CRelayM.I2CPacketMaster ->
           I2CPacketMasterC.I2CPacketMaster[CYCLOPS_I2C_ADDRESS];
```

```
    MoteI2CRelayM.I2CStdControl -> I2CPacketMasterC.StdControl;

}
```

B.2 MoteI2CRelayM.nc

```
// Modified by Paul Bender (bender.13@wright.edu) to send an
// image using a MICAz mote from a Cyclops camera

module MoteI2CRelayM {
    provides interface StdControl;
    uses {
        interface Leds;

        //RF communication
        interface StdControl as CommControl;
        interface SendMsg as SendMsg;
        interface ReceiveMsg as ReceiveMsg;

        //I2C communication
        interface I2CPacketMaster;
        interface StdControl as I2CStdControl;

        //Timer
        interface Timer;
    }
}
implementation {
    TOS_Msg msg1;             /* Message to be sent out */
    CyclopsPacket request;    /* Request to be sent to Cyclops*/

    command result_t StdControl.init() {
       //Initialize Radio stack
       call CommControl.init();
       //Initialize host-side of I2C
       call I2CStdControl.init();
       //Initialize leds
       call Leds.init();

       return SUCCESS;
    }
    command result_t StdControl.start() {
```

```
        //Start radio stack
        call CommControl.start();
        //Start host-side of I2C
        call I2CStdControl.start();

            /*****************************************************
             *PARAMETERS TO SET:
             * request.framesize
             *    Sets the output image size
             *    Possible values:
             *         32  <= 32x32 image
             *         64  <= 64x64 image
             *         128 <= 128x128 image
             * request.type
             *    Sets the type of output image
             *                      (color, black&white,ect)
             *    Possible values:
             *         CYCLOPS_IMAGE_TYPE_Y     <=Black and
             *                                    white image. 1
             *                                    byte per pixel.
             *         CYCLOPS_IMAGE_TYPE_RGB   <=Color image. 3
             *                                    bytes per pixel.
             *         CYCLOPS_IMAGE_TYPE_YCbCr <=Color image. 2
             *                                    bytes per pixel.
             *
             * NOTE: EXTENSIVE TESTING WAS NOT DONE WITH
             *       CYCLOPS_IMAGE_TYPE_YCbCr
             *****************************************************/
        request.frameSize = 128;
        request.type = CYCLOPS_IMAGE_TYPE_Y;

        //Start timer
        call Timer.start(TIMER_ONE_SHOT, 10000);

        return SUCCESS;
    }

    command result_t StdControl.stop() {
        //Stop the radio stack
        call CommControl.stop();
        //Stop host-side of I2C
        call I2CStdControl.stop();
        return SUCCESS;
    }
```

```
event result_t Timer.fired() {
    call Leds.greenOn();
    //Write packet to cyclops over I2C
    call I2CPacketMaster.writePacket(sizeof(request),
                                     (char*)&request);
    return SUCCESS;
}
/*************************Communication Stack************/
event result_t SendMsg.sendDone(TOS_MsgPtr sent,
                                result_t success) {
    serialDumpHeader_t* dataPtr =
                          (serialDumpHeader_t*)sent->data;

    call Leds.yellowToggle();

    //If at least one more full packet remains for the image
    if(dataPtr->seq >= (2*CYCLOPS_DATA)) {
        call I2CPacketMaster.readPacket(TOSH_DATA_LENGTH,
                                        msg1.data);
        return SUCCESS;
    }
    //If less than a full packet remains for the image
    else if (dataPtr->seq > (CYCLOPS_DATA)) {
        call I2CPacketMaster.readPacket(
                             (dataPtr->seq-CYCLOPS_DATA)+
                             sizeof(serialDumpHeader_t),
                             msg1.data);
    }
    //If we have finished sending this image, start a new one
    else {
        //call Timer.start(TIMER_ONE_SHOT,3000);
        call Timer.start(TIMER_ONE_SHOT,60000);
                                     // every 60 seconds - PAB
    }

    return SUCCESS;

}

event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr data) {
    return data;
}
/*****************************I2C*************************/
event result_t I2CPacketMaster.readPacketDone(char len,
                                              char *data) {
```

```
        call SendMsg.send(TOS_BCAST_ADDR,len,&msg1);
        return SUCCESS;
    }
    event result_t I2CPacketMaster.writePacketDone(bool r) {
        if(r == FAIL) {
            call Leds.redToggle();
            return FAIL;
        }
        //Read packet from I2C
        call I2CPacketMaster.readPacket(TOSH_DATA_LENGTH,
                                        msg1.data);
        return SUCCESS;
    }
}
```

Appendix C:Single Hop with out Camera - unmodified

C.1 MoteNoDataRelayC.nc

```
// Modified by Paul Bender (bender.13@wright.edu) to send a mote
// generated image instead of an image from a Cyclops Camera

includes MoteNoDataRelay;

configuration MoteNoDataRelayC { }
implementation {
    components
        Main,
        MoteNoDataRelayM,
        GenericComm as Comm,
        LedsC,
        TimerC;//, I2CPacketMasterC;

    Main.StdControl -> MoteNoDataRelayM;
    Main.StdControl -> TimerC;
    Main.StdControl -> Comm;


    MoteNoDataRelayM.Leds -> LedsC;
    MoteNoDataRelayM.Timer -> TimerC.Timer[unique("Timer")];

    //RF communication facility
    MoteNoDataRelayM.CommControl -> Comm;
    MoteNoDataRelayM.SendMsg -> Comm.SendMsg[Sample_Packet];
    MoteNoDataRelayM.ReceiveMsg ->Comm.ReceiveMsg[Sample_Packet];

    //Communcation over I2C
    //MoteI2CRelayM.I2CPacketMaster ->
```

```
    //     I2CPacketMasterC.I2CPacketMaster[CYCLOPS_I2C_ADDRESS];
    //MoteI2CRelayM.I2CStdControl -> I2CPacketMasterC.StdControl;


}
```

C.2 MoteNoDataRelayM.nc

```
// Modified by Paul Bender (bender.13@wright.edu) to send a mote
// generated image instead of an image from a Cyclops Camera

module MoteNoDataRelayM {
    provides interface StdControl;
    uses {
        interface Leds;

        //RF communication
        interface StdControl as CommControl;
        interface SendMsg as SendMsg;
        interface ReceiveMsg as ReceiveMsg;

        //I2C communication
        // interface I2CPacketMaster;
        // interface StdControl as I2CStdControl;

        //Timer
        interface Timer;
    }
}
implementation {
    TOS_Msg msg1;             /* Message to be sent out */
    CyclopsPacket request;    /* Request to be sent to Cyclops*/
    int sequence;
    int i;

    command result_t StdControl.init() {
       //Initialize Radio stack
       call CommControl.init();
       //Initialize host-side of I2C
       // call I2CStdControl.init();
       //Initialize leds
       call Leds.init();
```

```
   return SUCCESS;
}
command result_t StdControl.start() {
    //Start radio stack
    call CommControl.start();
    //Start host-side of I2C
    //call I2CStdControl.start();

        /*****************************************************
         *PARAMETERS TO SET:
         * request.framesize
         *    Sets the output image size
         *    Possible values:
         *         32  <= 32x32 image
         *         64  <= 64x64 image
         *         128 <= 128x128 image
         * request.type
         *    Sets the type of output image (color,
         *                                   black&white,ect)
         *    Possible values:
         *         CYCLOPS_IMAGE_TYPE_Y     <=Black and white
         *                                    image. 1 byte
         *                                    per pixel.
         *         CYCLOPS_IMAGE_TYPE_RGB   <=Color image. 3
         *                                    bytes per pixel.
         *         CYCLOPS_IMAGE_TYPE_YCbCr <=Color image. 2
         *                                    bytes per pixel.
         *
         * NOTE: EXTENSIVE TESTING WAS NOT DONE WITH
         *       CYCLOPS_IMAGE_TYPE_YCbCr
         *****************************************************/
    request.frameSize = 128;
    request.type = CYCLOPS_IMAGE_TYPE_Y;

    //Start timer
    call Timer.start(TIMER_ONE_SHOT, 10000);

    return SUCCESS;
}

command result_t StdControl.stop() {
    //Stop the radio stack
    call CommControl.stop();
    //Stop host-side of I2C
```

```
        // call I2CStdControl.stop();
        return SUCCESS;
    }

    event result_t Timer.fired() {
        serialDumpHeader_t header; // header for packet

        call Leds.greenOn();
        call Leds.redToggle();
        //Write packet to cyclops over I2C
        //call I2CPacketMaster.writePacket(sizeof(request),
        //                                 (char*)&request);
        sequence=128*128;
        header.seq=sequence;
        memcpy(msg1.data,&header,sizeof(serialDumpHeader_t));
        for(i=sizeof(serialDumpHeader_t);i<TOSH_DATA_LENGTH;i++)
            msg1.data[i]=127;
        call SendMsg.send(TOS_BCAST_ADDR,TOSH_DATA_LENGTH,&msg1);
        return SUCCESS;
    }

    /**************************Communication Stack*************/
    event result_t SendMsg.sendDone(TOS_MsgPtr sent,
                                    result_t success) {
        serialDumpHeader_t* dataPtr =
                                (serialDumpHeader_t*)sent->data;
        serialDumpHeader_t header; // header for packet
        call Leds.yellowToggle();
        //If at least one more full packet remains for the image
        if(dataPtr->seq >= (2*CYCLOPS_DATA)) {
            //call I2CPacketMaster.readPacket(TOSH_DATA_LENGTH,
            //                                  msg1.data);
            sequence=sequence - CYCLOPS_DATA;
            header.seq=sequence;
            memcpy(msg1.data,&header,sizeof(serialDumpHeader_t));
            for(i=sizeof(serialDumpHeader_t);i<TOSH_DATA_LENGTH;
                i++)
                msg1.data[i]=127;
            call SendMsg.send(TOS_BCAST_ADDR,TOSH_DATA_LENGTH,
                              &msg1);
            return SUCCESS;
        }
        //If less than a full packet remains for the image
        else if (dataPtr->seq > (CYCLOPS_DATA)) {
            //call I2CPacketMaster.readPacket(
```

```
            //                  (dataPtr->seq-CYCLOPS_DATA)+
            //                  sizeof(serialDumpHeader_t),
            //                  msg1.data);
            sequence=sequence - CYCLOPS_DATA;
            header.seq=sequence;
            memcpy(msg1.data,&header,sizeof(serialDumpHeader_t));
            for(i=sizeof(serialDumpHeader_t);
                i<=(sequence+sizeof(serialDumpHeader_t));i++)
                msg1.data[i]=127;
            call SendMsg.send(TOS_BCAST_ADDR,sequence+
                              sizeof(serialDumpHeader_t),&msg1);
            call Leds.redToggle();
            return SUCCESS;
        }
        //If we have finished sending this image, start a new one
        else {
            //call Timer.start(TIMER_ONE_SHOT,3000);
            call Timer.start(TIMER_ONE_SHOT,60000);  // every 60
                                                  //seconds - PAB
        }

        return SUCCESS;

    }

    event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr data) {
        return data;
    }
    /****************************I2C*****************************/
    //event result_t I2CPacketMaster.readPacketDone(char len,
    //                                              char *data) {
    //    call SendMsg.send(TOS_BCAST_ADDR,len,&msg1);
    //    return SUCCESS;
    //}
    //event result_t I2CPacketMaster.writePacketDone(bool r) {
    //    if(r == FAIL) {
    //        call Leds.redToggle();
    //        return FAIL;
    //    }
        //Read packet from I2C
    //    call I2CPacketMaster.readPacket(TOSH_DATA_LENGTH,
    //                                    msg1.data);
    //    return SUCCESS;
    //}
}
```

Appendix D:Single Hop Base - Mote

D.1 GenericBase.nc

```
// $Id: GenericBase.nc,v 1.1 2005/04/14 22:56:22 local Exp $

/*
 * "Copyright (c) 2000-2003 The Regents of the University  of
 * California.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for any purpose, without fee, and
 * without written agreement is hereby granted, provided that the
 * above copyright notice, the following two paragraphs and the
 * author appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO
 * ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR
 * CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE
 * AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA
 * HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS"
 * BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR
 * MODIFICATIONS."
 *
 * Copyright (c) 2002-2003 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached
 * INTEL-LICENSE file. If you do not find these files, copies can
```

```
 * be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300,
 * Berkeley, CA, 94704.  Attention:  Intel License Inquiry.
 */
configuration GenericBase {
}
implementation {
  components Main, GenericBaseM, RadioCRCPacket as Comm,
             UARTNoCRCPacket, LedsC;

  Main.StdControl -> GenericBaseM;

  GenericBaseM.UARTControl -> UARTNoCRCPacket;
  GenericBaseM.UARTSend -> UARTNoCRCPacket;
  GenericBaseM.UARTReceive -> UARTNoCRCPacket;

  GenericBaseM.RadioControl -> Comm;
  GenericBaseM.RadioSend -> Comm;
  GenericBaseM.RadioReceive -> Comm;

  GenericBaseM.Leds -> LedsC;
}
```

Appendix E:Single Hop with Camera with transmission Power Control

E.1 MoteI2CRelayC.nc

```
// Modified by Paul Bender (bender.13@wright.edu) to send an
// image using a MICAz mote from a Cyclops camera and to
// allow setting the transmission power.

includes MoteI2CRelay;

configuration MoteI2CRelayC { }
implementation {
    components
        Main,
        MoteI2CRelayM,
        GenericComm as Comm,
        LedsC,
        TimerC, I2CPacketMasterC ,
        CC2420RadioC;

    Main.StdControl -> MoteI2CRelayM;
    Main.StdControl -> TimerC;
    Main.StdControl -> Comm;

    MoteI2CRelayM.Leds -> LedsC;
    MoteI2CRelayM.Timer -> TimerC.Timer[unique("Timer")];

    //RF communication facility
    MoteI2CRelayM.CommControl -> Comm;
    MoteI2CRelayM.PowerControl -> CC2420RadioC;
    MoteI2CRelayM.SendMsg -> Comm.SendMsg[Sample_Packet];
    MoteI2CRelayM.ReceiveMsg -> Comm.ReceiveMsg[Sample_Packet];
```

```
    //Communcation over I2C
    MoteI2CRelayM.I2CPacketMaster ->
         I2CPacketMasterC.I2CPacketMaster[CYCLOPS_I2C_ADDRESS];
    MoteI2CRelayM.I2CStdControl -> I2CPacketMasterC.StdControl;

}
```

E.2 MoteI2CRelayM.nc

```
// Modified by Paul Bender (bender.13@wright.edu) to send an
// image using a MICAz mote from a Cyclops camera and to
// allow setting the transmission power.

module MoteI2CRelayM {
    provides interface StdControl;
    uses {
        interface Leds;

        //RF communication
        interface StdControl as CommControl;
        interface SendMsg as SendMsg;
        interface ReceiveMsg as ReceiveMsg;
        interface CC2420Control as PowerControl;

        //I2C communication
        interface I2CPacketMaster;
        interface StdControl as I2CStdControl;

        //Timer
        interface Timer;
    }
}
implementation {
    TOS_Msg msg1;             /* Message to be sent out */
    CyclopsPacket request;    /* Request to be sent to Cyclops*/

    command result_t StdControl.init() {
       //Initialize Radio stack
       call CommControl.init();
       //Initialize host-side of I2C
       call I2CStdControl.init();
       //Initialize leds
```

```
    call Leds.init();

    return SUCCESS;
  }
  command result_t StdControl.start() {
      //Start radio stack
      call CommControl.start();
     /* Set the RF Power. Argument between 3 (lowest) and
        31 (highest, default)*/
      call PowerControl.SetRFPower(11);
      /* set the RF channel to the desired channel.
         NOTE: Must match base
         NOTE: Chanel 25 and 26 do not interfere with 802.11
       */
      //call PowerControl.TunePreset(26);
      //Start host-side of I2C
      call I2CStdControl.start();

          /*****************************************************
           *PARAMETERS TO SET:
           * request.framesize
           *    Sets the output image size
           *    Possible values:
           *         32  <= 32x32 image
           *         64  <= 64x64 image
           *         128 <= 128x128 image
           * request.type
           *    Sets the type of output image (color,
           *                                   black&white,ect)
           *    Possible values:
           *         CYCLOPS_IMAGE_TYPE_Y     <= Black and
           *                                     white image. 1
           *                                     byte per pixel.
           *         CYCLOPS_IMAGE_TYPE_RGB   <= Color image. 3
           *                                     bytes per pixel.
           *         CYCLOPS_IMAGE_TYPE_YCbCr <= Color image. 2
           *                                     bytes per pixel.
           *
           * NOTE: EXTENSIVE TESTING WAS NOT DONE WITH
           *       CYCLOPS_IMAGE_TYPE_YCbCr
           *****************************************************/
      request.frameSize = 128;
      request.type = CYCLOPS_IMAGE_TYPE_Y;

      //Start timer
```

```
        call Timer.start(TIMER_ONE_SHOT, 10000);

        return SUCCESS;
    }

    command result_t StdControl.stop() {
        //Stop the radio stack
        call CommControl.stop();
        //Stop host-side of I2C
        call I2CStdControl.stop();
        return SUCCESS;
    }

    event result_t Timer.fired() {
        call Leds.greenOn();
        //Write packet to cyclops over I2C
        call I2CPacketMaster.writePacket(sizeof(request),
                                         (char*)&request);
        return SUCCESS;
    }
    /*************************Communication Stack*************/
    event result_t SendMsg.sendDone(TOS_MsgPtr sent,
                                    result_t success) {
        serialDumpHeader_t* dataPtr =
                                  (serialDumpHeader_t*)sent->data;

        call Leds.yellowToggle();

        //If at least one more full packet remains for the image
        if(dataPtr->seq >= (2*CYCLOPS_DATA)) {
            call I2CPacketMaster.readPacket(TOSH_DATA_LENGTH,
                                            msg1.data);
            return SUCCESS;
        }
        //If less than a full packet remains for the image
        else if (dataPtr->seq > (CYCLOPS_DATA)) {
            call I2CPacketMaster.readPacket(
                                   (dataPtr->seq-CYCLOPS_DATA)+
                                   sizeof(serialDumpHeader_t),
                                   msg1.data);
        }
        //If we have finished sending this image, start a new one
        else {
            //call Timer.start(TIMER_ONE_SHOT,3000);
            call Timer.start(TIMER_ONE_SHOT,5000);
```

```
                                                // every 5 seconds - PAB
        }

        return SUCCESS;

    }

    event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr pMsg) {
        return pMsg;
    }
    /*****************************I2C*****************************/
    event result_t I2CPacketMaster.readPacketDone(char len,
                                                  char *data) {
        call SendMsg.send(TOS_BCAST_ADDR,len,&msg1);
        return SUCCESS;
    }
    event result_t I2CPacketMaster.writePacketDone(bool r) {
        if(r == FAIL) {
            call Leds.redToggle();
            return FAIL;
        }
        //Read packet from I2C
        call I2CPacketMaster.readPacket(TOSH_DATA_LENGTH,
                                        msg1.data);
        return SUCCESS;
    }
}
```

Appendix F:Single Hop with out Camera with transmission Power Control

F.1 MoteNoDataRelayC.nc

```
// Modified by Paul Bender (bender.13@wright.edu) to send a mote
// generated image instead of an image from a Cyclops Camera and
// to allow setting the transmission power.

includes MoteNoDataRelay;

configuration MoteNoDataRelayC { }
implementation {
    components
        Main,
        MoteNoDataRelayM,
        GenericComm as Comm,
        LedsC,
        TimerC,
        CC2420RadioC;//, I2CPacketMasterC;

    Main.StdControl -> MoteNoDataRelayM;
    Main.StdControl -> TimerC;
    Main.StdControl -> Comm;


    MoteNoDataRelayM.Leds -> LedsC;
    MoteNoDataRelayM.Timer -> TimerC.Timer[unique("Timer")];

    //RF communication facility
    MoteNoDataRelayM.CommControl -> Comm;
    MoteNoDataRelayM.PowerControl -> CC2420RadioC;
    MoteNoDataRelayM.SendMsg -> Comm.SendMsg[Sample_Packet];
    MoteNoDataRelayM.ReceiveMsg ->
```

```
                                    Comm.ReceiveMsg[Sample_Packet];

    //Communcation over I2C
    //MoteI2CRelayM.I2CPacketMaster ->
    //     I2CPacketMasterC.I2CPacketMaster[CYCLOPS_I2C_ADDRESS];
    //MoteI2CRelayM.I2CStdControl -> I2CPacketMasterC.StdControl;


}
```

F.2 MoteNoDataRelayM.nc

```
// Modified by Paul Bender (bender.13@wright.edu) to send a mote
// generated image instead of an image from a Cyclops Camera and
// to allow setting the transmission power.

module MoteNoDataRelayM {
    provides interface StdControl;
    uses {
        interface Leds;

        //RF communication
        interface StdControl as CommControl;
        interface SendMsg as SendMsg;
        interface ReceiveMsg as ReceiveMsg;
        interface CC2420Control as PowerControl;

        //I2C communication
        // interface I2CPacketMaster;
        // interface StdControl as I2CStdControl;

        //Timer
        interface Timer;
    }
}
implementation {
    TOS_Msg msg1;             /* Message to be sent out */
    CyclopsPacket request;    /* Request to be sent to Cyclops*/
    int sequence;
    int i;

    command result_t StdControl.init() {
```

```
    //Initialize Radio stack
    call CommControl.init();
    //Initialize host-side of I2C
    // call I2CStdControl.init();
    //Initialize leds
    call Leds.init();

    return SUCCESS;
  }
  command result_t StdControl.start() {
     //Start radio stack
     call CommControl.start();
     /* Set the RF Power. Argument between 3 (lowest) and
       31 (highest, default)*/
     call PowerControl.SetRFPower(31);
     /* set the RF channel to the desired channel.
        NOTE: Must match base
        NOTE: Chanel 25 and 26 do not interfere with 802.11
      */
     //call PowerControl.TunePreset(26);

     //Start host-side of I2C
     //call I2CStdControl.start();

         /*****************************************************
          *PARAMETERS TO SET:
          * request.framesize
          *    Sets the output image size
          *    Possible values:
          *         32  <= 32x32 image
          *         64  <= 64x64 image
          *         128 <= 128x128 image
          * request.type
          *    Sets the type of output image
          *                       (color, black&white,ect)
          *    Possible values:
          *         CYCLOPS_IMAGE_TYPE_Y     <=Black and
          *                                    white image. 1
          *                                    byte per pixel.
          *         CYCLOPS_IMAGE_TYPE_RGB   <=Color image. 3
          *                                    bytes per pixel.
          *         CYCLOPS_IMAGE_TYPE_YCbCr <=Color image. 2
          *                                    bytes per pixel.
          *
          * NOTE: EXTENSIVE TESTING WAS NOT DONE WITH
```

```
            *       CYCLOPS_IMAGE_TYPE_YCbCr
            **************************************************/
        request.frameSize = 128;
        request.type = CYCLOPS_IMAGE_TYPE_Y;

        //Start timer
        call Timer.start(TIMER_ONE_SHOT, 10000);

        return SUCCESS;
    }

    command result_t StdControl.stop() {
        //Stop the radio stack
        call CommControl.stop();
        //Stop host-side of I2C
        // call I2CStdControl.stop();
        return SUCCESS;
    }

    event result_t Timer.fired() {
        serialDumpHeader_t header; // header for packet

        call Leds.greenOn();
        call Leds.redToggle();
        //Write packet to cyclops over I2C
        //call I2CPacketMaster.writePacket(sizeof(request),
                                           (char*)&request);
        sequence=128*128;
        header.seq=sequence;
        memcpy(msg1.data,&header,sizeof(serialDumpHeader_t));
        for(i=sizeof(serialDumpHeader_t);i<TOSH_DATA_LENGTH;i++)
            msg1.data[i]=127;
        call SendMsg.send(TOS_BCAST_ADDR,TOSH_DATA_LENGTH,&msg1);
        return SUCCESS;
    }

    /*************************Communication Stack************/
    event result_t SendMsg.sendDone(TOS_MsgPtr sent,
                                    result_t success) {
        serialDumpHeader_t* dataPtr =
                                  (serialDumpHeader_t*)sent->data;
        serialDumpHeader_t header; // header for packet
        call Leds.yellowToggle();
        //If at least one more full packet remains for the image
        if(dataPtr->seq >= (2*CYCLOPS_DATA)) {
```

```
            //call I2CPacketMaster.readPacket(TOSH_DATA_LENGTH,
                                                msg1.data);
            sequence=sequence - CYCLOPS_DATA;
            header.seq=sequence;
            memcpy(msg1.data,&header,sizeof(serialDumpHeader_t));
            for(i=sizeof(serialDumpHeader_t);i<TOSH_DATA_LENGTH;
                        i++)
msg1.data[i]=127;
            call SendMsg.send(TOS_BCAST_ADDR,TOSH_DATA_LENGTH,
                              &msg1);
            return SUCCESS;
        }
        //If less than a full packet remains for the image
        else if (dataPtr->seq > (CYCLOPS_DATA)) {
            //call I2CPacketMaster.readPacket(
            //     (dataPtr->seq-CYCLOPS_DATA)+
            //     sizeof(serialDumpHeader_t), msg1.data);
            sequence=sequence - CYCLOPS_DATA;
            header.seq=sequence;
            memcpy(msg1.data,&header,sizeof(serialDumpHeader_t));
            for(i=sizeof(serialDumpHeader_t);
                i<=(sequence+sizeof(serialDumpHeader_t));i++)
msg1.data[i]=127;
            call SendMsg.send(TOS_BCAST_ADDR,sequence+
                              sizeof(serialDumpHeader_t),&msg1);
            call Leds.redToggle();
            return SUCCESS;
        }
        //If we have finished sending this image, start a new one
        else {
            //call Timer.start(TIMER_ONE_SHOT,3000);
            call Timer.start(TIMER_ONE_SHOT,5000);
                                        // every 5 seconds - PAB
        }

        return SUCCESS;

    }

    event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr data) {
        return data;
    }
    /*****************************I2C*****************************/
    //event result_t I2CPacketMaster.readPacketDone(char len,
                                                    char *data) {
```

```
    //    call SendMsg.send(TOS_BCAST_ADDR,len,&msg1);
    //    return SUCCESS;
    //}
    //event result_t I2CPacketMaster.writePacketDone(bool r) {
    //    if(r == FAIL) {
    //        call Leds.redToggle();
    //        return FAIL;
    //    }
        //Read packet from I2C
    //    call I2CPacketMaster.readPacket(TOSH_DATA_LENGTH,
    //                                    msg1.data);
    //    return SUCCESS;
    //}
}
```

Appendix G:Multi-Hop Mote Code

G.1 MoteI2CRelayC.nc

```
includes MoteI2CRelay;
includes CyclopsNetwork;
includes MultiHop;

// Modified by Paul Bender (bender.13@wright.edu) to use MICAz
// motesto send Cyclops images over multiple hops to the sink
// node using the reliableroute module.

configuration MoteI2CRelayC { }
implementation {
    components
        Main,
        MoteI2CRelayM,
        CC2420RadioC,
        TimerC, I2CPacketMasterC,
        Bcast,
        GenericCommPromiscuous as Comm,
        //WMEWMAMultiHopRouter as multihopM,
        EWMAMultiHopRouter as multihopM,
        //MultiHopRouter as multihopM,
        QueuedSend,
        LedsC;

    Main.StdControl -> MoteI2CRelayM;
    Main.StdControl -> TimerC;
    Main.StdControl -> Bcast.StdControl;
    Main.StdControl -> multihopM.StdControl;
    Main.StdControl -> QueuedSend.StdControl;
    Main.StdControl -> Comm;

    MoteI2CRelayM.Leds -> LedsC;
    MoteI2CRelayM.Timer -> TimerC.Timer[unique("Timer")];
```

```
    //MoteI2CRelayM.Timer2 -> TimerC.Timer[unique("Timer2")];

    //RF communication facility
    //MoteI2CRelayM.CommControl -> Comm;
    MoteI2CRelayM.PowerControl -> CC2420RadioC;
    //MoteI2CRelayM.SendMsg -> Comm.SendMsg[Sample_Packet];
    //MoteI2CRelayM.ReceiveMsg -> Comm.ReceiveMsg[Sample_Packet];


    MoteI2CRelayM.Bcast -> Bcast.Receive[AM_CYCLOPSMHOPCMDMSG];
    Bcast.ReceiveMsg[AM_CYCLOPSMHOPCMDMSG] ->
                        Comm.ReceiveMsg[AM_CYCLOPSMHOPCMDMSG];

    multihopM.ReceiveMsg[AM_CYCLOPSMHOPCMDMSG] ->
                        Comm.ReceiveMsg[AM_CYCLOPSMHOPCMDMSG];


    MoteI2CRelayM.RouteControl -> multihopM;
    MoteI2CRelayM.SendMsg ->
                        multihopM.Send[AM_CYCLOPSMHOPDATAMSG];
    multihopM.ReceiveMsg[AM_CYCLOPSMHOPDATAMSG] ->
                        Comm.ReceiveMsg[AM_CYCLOPSMHOPDATAMSG];

    //Communcation over I2C
    MoteI2CRelayM.I2CPacketMaster ->
          I2CPacketMasterC.I2CPacketMaster[CYCLOPS_I2C_ADDRESS];
    MoteI2CRelayM.I2CStdControl -> I2CPacketMasterC.StdControl;

}
```

G.2 MoteI2CRelayM.nc

```
// Modified by Paul Bender (bender.13@wright.edu) to use MICAz
// motesto send Cyclops images over multiple hops to the sink
// node using the reliableroute module.

module MoteI2CRelayM {
    provides interface StdControl;
    //provides command result_t getNextPacket(uint8_t offset);
    uses {
        interface Leds;
```

```
        //RF communication
        interface StdControl as CommControl;
        //interface SendMsg as SendMsg;
        //interface ReceiveMsg as ReceiveMsg;
        interface Send as SendMsg;
        interface CC2420Control as PowerControl;
        interface Receive as Bcast;
        interface RouteControl;

        //I2C communication
        interface I2CPacketMaster;
        interface StdControl as I2CStdControl;

        //Timer
        interface Timer;
        //interface Timer as Timer2;
    }
}
implementation {
    TOS_Msg msg1;             /* Message to be sent out */
    uint16_t length;               /* length of the payload */
    CyclopsPacket request;    /* Request to be sent to Cyclops*/
    bool CameraOn = TRUE;
    bool TimerRunning = FALSE;
    int sequence;

    command result_t StdControl.init() {
       //Initialize Radio stack
       //call CommControl.init();
       //Initialize host-side of I2C
       call I2CStdControl.init();
       //Initialize leds
       call Leds.init();

       return SUCCESS;
    }
    command result_t StdControl.start() {
        //Start radio stack
        //call CommControl.start();
       /* Set the RF Power. Argument between 3 (lowest) and
          31 (highest, default)*/
        call PowerControl.SetRFPower(3);
        /* set the RF channel to the desired channel.
           NOTE: Must match base
           NOTE: Chanel 25 and 26 do not interfere with 802.11
```

```
         */
        //call PowerControl.TunePreset(26);
        /* turn automatic acknowledgements on */
        //call PowerControl.enableAutoAck();

        // set the update frequency
        call RouteControl.setUpdateInterval(30);

        //Start host-side of I2C
        call I2CStdControl.start();

            /*****************************************************
             *PARAMETERS TO SET:
             * request.framesize
             *    Sets the output image size
             *    Possible values:
             *         32  <= 32x32 image
             *         64  <= 64x64 image
             *         128 <= 128x128 image
             * request.type
             *    Sets the type of output image (color,
             *                                   black&white,ect)
             *    Possible values:
             *         CYCLOPS_IMAGE_TYPE_Y     <= Black and
             *                                     white image. 1
             *                                     byte per pixel.
             *         CYCLOPS_IMAGE_TYPE_RGB   <= Color image. 3
             *                                     bytes per pixel.
             *         CYCLOPS_IMAGE_TYPE_YCbCr <= Color image. 2
             *                                     bytes per pixel.
             *
             * NOTE: EXTENSIVE TESTING WAS NOT DONE WITH
             *       CYCLOPS_IMAGE_TYPE_YCbCr
             *****************************************************/
        request.frameSize = 128;
        request.type = CYCLOPS_IMAGE_TYPE_Y;

        //Start timer
        TimerRunning=FALSE;
        call Timer.start(TIMER_ONE_SHOT, 10000);

        return SUCCESS;
    }

    command result_t StdControl.stop() {
```

```
        //Stop the radio stack
        //call CommControl.stop();
        //Stop host-side of I2C
        call I2CStdControl.stop();
        return SUCCESS;
    }

    event result_t Timer.fired() {
        uint16_t parent;
        call Leds.greenOn();
        if((parent=call RouteControl.getParent())==0xffff)
{
           // manually trigger a routing update
           call RouteControl.manualUpdate();
           call Leds.greenOff();
        }
        else {
        //Write packet to cyclops over I2C
        sequence=128*128;
        if((TimerRunning==FALSE))
{
           TimerRunning=TRUE;
           call Timer.start(TIMER_REPEAT,60000);
                                // every 60 seconds - PAB
}
        //call Timer.start(TIMER_ONE_SHOT, 10000);
        call I2CPacketMaster.writePacket(sizeof(request),
                                (char*)&request);
        }
        return SUCCESS;
    }

    /*************************Communication Stack*************/
    //command result_t getNextPacket(uint8_t offset) {
    task void getNextPacket() {
        uint8_t *pData;

        call Leds.yellowToggle();

        //If at least one more full packet remains for the image
        if(sequence >= (2*CYCLOPS_DATA)) {
            if ((pData =
               (uint8_t *)call SendMsg.getBuffer(&msg1,&length)))
            {
               call I2CPacketMaster.readPacket(length,pData);
```

```
            }
        }
        //If less than a full packet remains for the image
        else if (sequence > (CYCLOPS_DATA)) {
           if ((pData =
              (uint8_t *)call SendMsg.getBuffer(&msg1,&length)))
           {
               call I2CPacketMaster.readPacket(
                                (sequence-CYCLOPS_DATA)+
                                sizeof(serialDumpHeader_t),
                                pData);
           }
        }
        sequence=sequence-CYCLOPS_DATA;
    }

    /*event result_t Timer2.fired() {
        post getNextPacket();
        return SUCCESS;
    }*/

    task void sendData()
    {
       call SendMsg.send(&msg1,length);
    }

    event result_t SendMsg.sendDone(TOS_MsgPtr sent,
                                    result_t success) {
        //serialDumpHeader_t* dataPtr =
        //                      (serialDumpHeader_t*)sent->data;
TOSH_uwait(10000);
post getNextPacket();
        //call Timer2.start(TIMER_ONE_SHOT, 100);

        return SUCCESS;
    }

    event TOS_MsgPtr Bcast.receive(TOS_MsgPtr pMsg,
                                   void* payload,
                                   uint16_t payloadLen){

    CyclopsCmdMsg *pCmdMsg = (CyclopsCmdMsg *)payload;

    if (pCmdMsg->type == CYCLOPS_TYPE_CAMERAOFF ) {
      CameraOn = FALSE ;
```

```
      // Need to implement, send turn power off to camera
      call Timer.stop();

    } else if (pCmdMsg->type == CYCLOPS_TYPE_CAMERAON) {

        // Need to impelment, send turn power on to camera
        CameraOn = TRUE;
    } else if (pCmdMsg->type == CYCLOPS_TYPE_SETSIZE ) {
      // Set the size of the image
      request.frameSize=pCmdMsg->args.imagesize;
    } else if (pCmdMsg->type == CYCLOPS_TYPE_SETTYPE ) {
      // Set the type of the image
      request.type = pCmdMsg->args.imagetype;
    }
        return pMsg;
    }

    //event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr pMsg) {
    //    return pMsg;
    //}
    /*****************************I2C*****************************/
    event result_t I2CPacketMaster.readPacketDone(char len,
                                                  char *data) {
        //call SendMsg.send(TOS_BCAST_ADDR,len,&msg1);
post sendData();
        return SUCCESS;
    }
    event result_t I2CPacketMaster.writePacketDone(bool r) {
        uint8_t *pData;
        if(r == FAIL) {
            call Leds.redToggle();
            return FAIL;
        }
        //Read packet from I2C
        if ((pData = (uint8_t *)call SendMsg.getBuffer(&msg1,
                                                       &length))){
           call I2CPacketMaster.readPacket(length, pData);
           return SUCCESS;
        }
        else return FAIL;
   }
}
```

G.3 CyclopsNetwork.h

```
#include "AM.h"

// New File created by Paul Bender (bender.13@wright.edu)
// This file defines the message formats for a multihop
// network which sends data from a cyclops camera to a sink
// using MICAz motes and a TinyOS 1.x routing protocol.

enum {
  CYCLOPS_TYPE_SENSORREADING = 0,
  CYCLOPS_TYPE_SETSIZE= 1,
  CYCLOPS_TYPE_SETTYPE = 2,
  CYCLOPS_TYPE_CAMERAON= 3,
  CYCLOPS_TYPE_CAMERAOFF = 4,
};

typedef struct CyclopsCmdMsg{
  uint8_t type;
  uint16_t parentaddr;
  union {
    // FOR CYCLOPS_TYPE_SETSIZE
    uint8_t imagesize;
    // FOR CYCLOPS_TYPE_SETTYPE
    uint8_t imagetype;
  } args;

} __attribute__ ((packed)) CyclopsCmdMsg;

enum {
  AM_CYCLOPSMHOPCMDMSG=24
};

typedef struct CyclopsMHopDataMsg {
  uint8_t data[29];
} __attribute__ ((packed)) CyclopsMHopDataMsg;

enum {
  AM_CYCLOPSMHOPDATAMSG=23
};
```

Appendix H:Multi-Hop Base - Mote

H.1 MultiHopBase.nc

```
// $Id: GenericBase.nc,v 1.1 2005/04/14 22:56:22 local Exp $

/*
 * "Copyright (c) 2000-2003 The Regents of the University  of
 * California.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for any purpose, without fee, and
 * without written agreement is hereby granted, provided that the
 * above copyright notice, the following two paragraphs and the
 * author appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO
 * ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR
 * CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE
 * AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA
 * HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS"
 * BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR
 * MODIFICATIONS."
 *
 * Copyright (c) 2002-2003 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached
 * INTEL-LICENSE file. If you do not find these files, copies can
```

```
 * be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300,
 * Berkeley, CA, 94704.  Attention:  Intel License Inquiry.
 */

 /* Modified by Paul Bender (bender.13@wright.edu) to utilize
  * multihop routing.
  */

includes CyclopsNetwork;
includes MultiHop;

configuration MultihopBase {
}
implementation {
  components Main,
  MultihopBaseM,
  RadioCRCPacket as Comm,
  //GenericCommPromiscuous as Comm,
  //WMEWMAMultiHopRouter as multihopM,
  EWMAMultiHopRouter as multihopM,
  QueuedSend,
  UARTNoCRCPacket, LedsC,
  TimerC,
  CC2420RadioC;

  Main.StdControl -> MultihopBaseM;
  Main.StdControl -> multihopM.StdControl;
  Main.StdControl -> QueuedSend.StdControl;
  Main.StdControl -> Comm;
  Main.StdControl -> TimerC;

  MultihopBaseM.UARTControl -> UARTNoCRCPacket;
  MultihopBaseM.UARTSend -> UARTNoCRCPacket;
  MultihopBaseM.UARTReceive -> UARTNoCRCPacket;

  MultihopBaseM.RadioControl -> Comm;
  MultihopBaseM.RadioSend -> Comm;
  MultihopBaseM.RadioReceive -> Comm;

  //MultihopBaseM.RadioSend ->
  //                        multihopM.Send[AM_CYCLOPSMHOPCMDMSG];

  //MultihopBaseM.RadioReceive->
  //                    multihopM.Receive[AM_CYCLOPSMHOPDATAMSG];
```

```
  //MultihopBaseM.RadioReceive->
  //                            multihopM.Receive[AM_MULTIHOPMSG];
  //MultihopBaseM.RadioReceive->
  //                            multihopM.Receive[AM_DEBUGPACKET];

  MultihopBaseM.PowerControl -> CC2420RadioC;

  MultihopBaseM.Leds -> LedsC;

  MultihopBaseM.RouteControl -> multihopM;

  MultihopBaseM.Timer -> TimerC.Timer[unique("Timer")];
}
```

H.2 MultiHopBaseM.nc

```
// $Id: GenericBaseM.nc,v 1.1 2005/04/14 22:56:22 local Exp $
/*
 * "Copyright (c) 2000-2003 The Regents of the University  of
 * California.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software
 * and its documentation for any purpose, without fee, and
 * without written agreement is hereby granted, provided that the
 * above copyright notice, the following two paragraphs and the
 * author appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO
 * ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR
 * CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE
 * AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA
 * HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY
 * WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
 * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS"
 * BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR
 * MODIFICATIONS."
 *
 * Copyright (c) 2002-2003 Intel Corporation
```

```
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached
 * INTEL-LICENSE file. If you do not find these files, copies can
 * be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300,
 * Berkeley, CA, 94704.  Attention:  Intel License Inquiry.
 */


/* History:   created 1/25/2001
 *
 *
 * Modified by Paul Bender (bender.13@wright.edu) to utilize
 * multihop routing.
 */

/* Generic.Base.c
   - captures all the packets that it can hear and report it back
     to the UART
   - forward all incoming UART messages out to the radio
*/

module MultihopBaseM{
  provides interface StdControl;
  uses {
    interface StdControl as UARTControl;
    interface BareSendMsg as UARTSend;
    interface ReceiveMsg as UARTReceive;

    interface StdControl as RadioControl;
    interface BareSendMsg as RadioSend;
    //interface Send as RadioSend;
    interface ReceiveMsg as RadioReceive;
    //interface Receive as RadioReceive;

    interface CC2420Control as PowerControl;
    interface RouteControl;

    interface Leds;

    interface Timer;
  }
}
implementation
```

```
{
  TOS_Msg buffer;
  TOS_MsgPtr ourBuffer;
  bool sendPending;


  /* Generic.Base.Init:
     initialize lower components.
     initialize component state, including constant portion of
     msgs.
  */
  command result_t StdControl.init() {
    result_t ok1, ok2, ok3;

    ourBuffer = &buffer;
    sendPending = TRUE;

    ok1 = call UARTControl.init();
    ok2 = call RadioControl.init();
    ok3 = call Leds.init();

    //call PowerControl.enableAutoAck();

    sendPending = FALSE;

    dbg(DBG_BOOT, "GenericBase initialized\n");

    return rcombine3(ok1, ok2, ok3);
  }

  command result_t StdControl.start() {
    result_t ok1, ok2;

    ok1 = call UARTControl.start();
    ok2 = call RadioControl.start();

    // manually trigger a routing update
    call RouteControl.manualUpdate();

    call Timer.start(TIMER_REPEAT,60000);
                              // every 60 seconds - PAB
    return rcombine(ok1, ok2);
  }
```

```
 command result_t StdControl.stop() {
    result_t ok1, ok2;

    ok1 = call UARTControl.stop();
    ok2 = call RadioControl.stop();

    return rcombine(ok1, ok2);
  }

  TOS_MsgPtr receive(TOS_MsgPtr received, bool fromUART) {
    TOS_MsgPtr nextReceiveBuffer = received;

    dbg(DBG_USR1, "GenericBase received %s packet\n",
fromUART ? "UART" : "radio");
    if ((!sendPending) &&
(received->group == (TOS_AM_GROUP & 0xff))) {

      result_t ok;
      nextReceiveBuffer = ourBuffer;
      ourBuffer = received;
      dbg(DBG_USR1, "GenericBase forwarding packet to %s\n",
  fromUART ? "radio" : "UART");
      if (fromUART)
{
  call Leds.redToggle();
  ok = call RadioSend.send(received);
  //ok = call RadioSend.send(received,36);
}
else
  {
    call Leds.greenToggle();
    received->addr = TOS_UART_ADDR;
    ok = call UARTSend.send(received);
  }
      if (ok != FAIL)
{
  dbg(DBG_USR1, "GenericBase send pending\n");
  sendPending = TRUE;
}
      else {
call Leds.yellowToggle();
      }

    }
    // set the frequency of routing updates
```

```
    call RouteControl.setUpdateInterval(10);

    return nextReceiveBuffer;
  }

  result_t sendDone(TOS_MsgPtr sent, result_t success) {
    if(ourBuffer == sent)
      {
dbg(DBG_USR1, "GenericBase send buffer free\n");
if (success == FAIL)
  call Leds.yellowToggle();
sendPending = FALSE;
      }
    return SUCCESS;
  }


  event TOS_MsgPtr RadioReceive.receive(TOS_MsgPtr data) {
//event TOS_MsgPtr RadioReceive.receive(TOS_MsgPtr data,
//                    void* payload, uint16_t payloadLen){
    if (data->crc) {
      return receive(data, FALSE);
    }
    else {
      return data;
    }
  }

  event TOS_MsgPtr UARTReceive.receive(TOS_MsgPtr data) {
    return receive(data, TRUE);
  }

  event result_t UARTSend.sendDone(TOS_MsgPtr msg,
                                   result_t success) {
    return sendDone(msg, success);
  }

  event result_t RadioSend.sendDone(TOS_MsgPtr msg,
                                   result_t success) {
    return sendDone(msg, success);
  }

  event result_t Timer.fired() {
     // manually trigger a routing update
     call RouteControl.manualUpdate();
```

```
    }

}
```

Appendix I:Multi-Hop Reception Program - Host Computer

I.1 frame.c

```
//$Header: /home/cvs/cvsroot/tos-contrib/cyclops/tools/R/frame.c,
v 1.2 2005/04/19 23:30:44 local Exp $
//Modified by: Mohammad Rahimi mhr@cens.ucla.edu
//Modified by Paul Bender bender.13@wright.edu to work with MICAz
//                        using TinyOS multihop routing.

#include "serial.h"
#include "serialDump.h"

typedef struct HEADER{
    unsigned short int type;             /* Magic identifier */
    unsigned int size;                   /* File size in bytes */
    unsigned short int reserved1, reserved2;
    unsigned int offset;                 /* Offset to image data,
                                            bytes */
}  __attribute__((packed)) HEADER;
typedef struct INFOHEADER {
    unsigned int size;              /* Header size in bytes   */
    int width,height;               /* Width and height of image*/
    unsigned short int planes;      /* Number of colour planes */
    unsigned short int bits;        /* Bits per pixel */
    unsigned int compression;       /* Compression type */
    unsigned int imagesize;         /* Image size in bytes */
    int xresolution,yresolution;    /* Pixels per meter */
    unsigned int ncolours;          /* Number of colours  */
    unsigned int importantcolours;   /* Important colours  */
} INFOHEADER;

//**********************************************************
//This function grabs a packet
```

```
//*************************************************************
TOS_SERIAL_Msg read_packet()
{
    int count;
    char c;
    char input_buffer[sizeof(TOS_SERIAL_Msg)];
    TOS_SERIAL_Msg msg;
    int datasize=0;
    bzero(input_buffer,sizeof(TOS_SERIAL_Msg));

    //search through to find 0x7e signifing the start of a
    //packet
    while(input_buffer[0] != (char)(0x7e)){
        while((c = read(input_stream, input_buffer, 1)) != 1){
        };
    }
    count = 1;
    //you have the first byte now read the rest.
    while(count < 14 ){// sizeof(TOS_SERIAL_Msg)) {
        count += read(input_stream, input_buffer + count,1);
    }
    serialDumpHeader_t myHeader;
    memcpy(&myHeader,input_buffer+13,sizeof(serialDumpHeader_t));
    datasize=myHeader.seq>TOSH_DATA_LENGTH?
        TOSH_DATA_LENGTH-sizeof(serialDumpHeader_t):myHeader.seq;
    while(count < (datasize-1+(sizeof(TOS_SERIAL_Msg)-
                (TOSH_DATA_LENGTH-sizeof(serialDumpHeader_t))))){
        count += read(input_stream, input_buffer + count,1);
    }

    memcpy(&msg,input_buffer,sizeof(TOS_SERIAL_Msg));
    return msg;
}

//*************************************************************
//This function prints a packet
//*************************************************************
void print_packet(TOS_SERIAL_Msg myMsg)
{
    int i;
    char input_buffer[sizeof(TOS_SERIAL_Msg)];
    memcpy(input_buffer,&myMsg,sizeof(TOS_SERIAL_Msg));
    for(i=0; i <sizeof(TOS_SERIAL_Msg); i++)
                printf("%02x ", (unsigned char) input_buffer[i]);
    printf("\n");fflush(stdout);
```

```
    return;
}
void print_data(TOS_SERIAL_Msg myMsg)
{
    int i;
    char input_buffer[29];
    memcpy(input_buffer,&myMsg.data,29);
    for(i=0; i <29; i++)
       printf("%02x ", (unsigned char) input_buffer[i]);
    printf("\n");fflush(stdout);
    return;
}
//*************************************************************
//This function prints a packet
//*************************************************************
void print_dump(char *myBuf,int myBufSize)
{
    int i;
    printf("\n\n***************serial dump***************\n");
    for(i=0;i<myBufSize; i++)
        {
            if(i%16==0) printf("\n");
            printf("%2x ", (unsigned char) myBuf[i]);
        }
    printf("\n*******************end*******************\n");
    fflush(stdout);
    return;
}

//*************************************************************
//This function prints a packet
//*************************************************************
void print_file_dump(char *myBuf,int myBufSize)
{
    int i;
    FILE *fp1,*fp2;
    if((fp1 = fopen ("frame.dat", "w"))==NULL)
        printf("Data file error\n");
  //printf("\n\n***************serial dump***************\n");
    for(i=0;i<myBufSize; i++)
        {
            if(i%16==0 && i!=0) fprintf(fp1,"\n");
            fprintf(fp1,"%2x ", (unsigned char) myBuf[i]);
        }
    //printf("\n*******************end*******************\n");
```

```
    //fflush(stdout);
    fclose(fp1);
    //we only create lock file
    fp2 = fopen (".frame.Lock", "w");
    fclose(fp2);
    return;
}

//*************************************************************
//This function creates a bmp file of the image
//*************************************************************
void print_bmp_dump(char *myBuf, int myBufSize){
    FILE *fp;

    HEADER h1;
    INFOHEADER h2;
    //char *outputBuf;
    unsigned char palette[] = { \
                               0x00, 0x00, 0x00, 0x00, 0x01, 0x01,\
                               0x01, 0x00, 0x02, 0x02, 0x02, 0x00,\
                               0x03, 0x03, 0x03, 0x00, 0x04, 0x04,\
                               0x04, 0x00, 0x05, 0x05, 0x05, 0x00,\
                               0x06, 0x06, 0x06, 0x00, 0x07, 0x07,\
                               0x07, 0x00, 0x08, 0x08, 0x08, 0x00,\
                               0x09, 0x09, 0x09, 0x00, 0x0a, 0x0a,\
                               0x0a, 0x00, 0x0b, 0x0b, 0x0b, 0x00,\
                               0x0c, 0x0c, 0x0c, 0x00, 0x0d, 0x0d,\
                               0x0d, 0x00, 0x0e, 0x0e, 0x0e, 0x00,\
                               0x0f, 0x0f, 0x0f, 0x00, 0x10, 0x10,\
                               0x10, 0x00, 0x11, 0x11, 0x11, 0x00,\
                               0x12, 0x12, 0x12, 0x00, 0x13, 0x13,\
                               0x13, 0x00, 0x14, 0x14, 0x14, 0x00,\
                               0x15, 0x15, 0x15, 0x00, 0x16, 0x16,\
                               0x16, 0x00, 0x17, 0x17, 0x17, 0x00,\
                               0x18, 0x18, 0x18, 0x00, 0x19, 0x19,\
                               0x19, 0x00, 0x1a, 0x1a, 0x1a, 0x00,\
                               0x1b, 0x1b, 0x1b, 0x00, 0x1c, 0x1c,\
                               0x1c, 0x00, 0x1d, 0x1d, 0x1d, 0x00,\
                               0x1e, 0x1e, 0x1e, 0x00, 0x1f, 0x1f,\
                               0x1f, 0x00, 0x20, 0x20, 0x20, 0x00,\
                               0x21, 0x21, 0x21, 0x00, 0x22, 0x22,\
                               0x22, 0x00, 0x23, 0x23, 0x23, 0x00,\
                               0x24, 0x24, 0x24, 0x00, 0x25, 0x25,\
                               0x25, 0x00, 0x26, 0x26, 0x26, 0x00,\
                               0x27, 0x27, 0x27, 0x00, 0x28, 0x28,\
```

```
0x28, 0x00, 0x29, 0x29, 0x29, 0x00,\
0x2a, 0x2a, 0x2a, 0x00, 0x2b, 0x2b,\
0x2b, 0x00, 0x2c, 0x2c, 0x2c, 0x00,\
0x2d, 0x2d, 0x2d, 0x00, 0x2e, 0x2e,\
0x2e, 0x00, 0x2f, 0x2f, 0x2f, 0x00,\
0x30, 0x30, 0x30, 0x00, 0x31, 0x31,\
0x31, 0x00, 0x32, 0x32, 0x32, 0x00,\
0x33, 0x33, 0x33, 0x00, 0x34, 0x34,\
0x34, 0x00, 0x35, 0x35, 0x35, 0x00,\
0x36, 0x36, 0x36, 0x00, 0x37, 0x37,\
0x37, 0x00, 0x38, 0x38, 0x38, 0x00,\
0x39, 0x39, 0x39, 0x00, 0x3a, 0x3a,\
0x3a, 0x00, 0x3b, 0x3b, 0x3b, 0x00,\
0x3c, 0x3c, 0x3c, 0x00, 0x3d, 0x3d,\
0x3d, 0x00, 0x3e, 0x3e, 0x3e, 0x00,\
0x3f, 0x3f, 0x3f, 0x00, 0x40, 0x40,\
0x40, 0x00, 0x41, 0x41, 0x41, 0x00,\
0x42, 0x42, 0x42, 0x00, 0x43, 0x43,\
0x43, 0x00, 0x44, 0x44, 0x44, 0x00,\
0x45, 0x45, 0x45, 0x00, 0x46, 0x46,\
0x46, 0x00, 0x47, 0x47, 0x47, 0x00,\
0x48, 0x48, 0x48, 0x00, 0x49, 0x49,\
0x49, 0x00, 0x4a, 0x4a, 0x4a, 0x00,\
0x4b, 0x4b, 0x4b, 0x00, 0x4c, 0x4c,\
0x4c, 0x00, 0x4d, 0x4d, 0x4d, 0x00,\
0x4e, 0x4e, 0x4e, 0x00, 0x4f, 0x4f,\
0x4f, 0x00, 0x50, 0x50, 0x50, 0x00,\
0x51, 0x51, 0x51, 0x00, 0x52, 0x52,\
0x52, 0x00, 0x53, 0x53, 0x53, 0x00,\
0x54, 0x54, 0x54, 0x00, 0x55, 0x55,\
0x55, 0x00, 0x56, 0x56, 0x56, 0x00,\
0x57, 0x57, 0x57, 0x00, 0x58, 0x58,\
0x58, 0x00, 0x59, 0x59, 0x59, 0x00,\
0x5a, 0x5a, 0x5a, 0x00, 0x5b, 0x5b,\
0x5b, 0x00, 0x5c, 0x5c, 0x5c, 0x00,\
0x5d, 0x5d, 0x5d, 0x00, 0x5e, 0x5e,\
0x5e, 0x00, 0x5f, 0x5f, 0x5f, 0x00,\
0x60, 0x60, 0x60, 0x00, 0x61, 0x61,\
0x61, 0x00, 0x62, 0x62, 0x62, 0x00,\
0x63, 0x63, 0x63, 0x00, 0x64, 0x64,\
0x64, 0x00, 0x65, 0x65, 0x65, 0x00,\
0x66, 0x66, 0x66, 0x00, 0x67, 0x67,\
0x67, 0x00, 0x68, 0x68, 0x68, 0x00,\
0x69, 0x69, 0x69, 0x00, 0x6a, 0x6a,\
0x6a, 0x00, 0x6b, 0x6b, 0x6b, 0x00,\
```

```
0x6c, 0x6c, 0x6c, 0x00, 0x6d, 0x6d,\
0x6d, 0x00, 0x6e, 0x6e, 0x6e, 0x00,\
0x6f, 0x6f, 0x6f, 0x00, 0x70, 0x70,\
0x70, 0x00, 0x71, 0x71, 0x71, 0x00,\
0x72, 0x72, 0x72, 0x00, 0x73, 0x73,\
0x73, 0x00, 0x74, 0x74, 0x74, 0x00,\
0x75, 0x75, 0x75, 0x00, 0x76, 0x76,\
0x76, 0x00, 0x77, 0x77, 0x77, 0x00,\
0x78, 0x78, 0x78, 0x00, 0x79, 0x79,\
0x79, 0x00, 0x7a, 0x7a, 0x7a, 0x00,\
0x7b, 0x7b, 0x7b, 0x00, 0x7c, 0x7c,\
0x7c, 0x00, 0x7d, 0x7d, 0x7d, 0x00,\
0x7e, 0x7e, 0x7e, 0x00, 0x7f, 0x7f,\
0x7f, 0x00, 0x80, 0x80, 0x80, 0x00,\
0x81, 0x81, 0x81, 0x00, 0x82, 0x82,\
0x82, 0x00, 0x83, 0x83, 0x83, 0x00,\
0x84, 0x84, 0x84, 0x00, 0x85, 0x85,\
0x85, 0x00, 0x86, 0x86, 0x86, 0x00,\
0x87, 0x87, 0x87, 0x00, 0x88, 0x88,\
0x88, 0x00, 0x89, 0x89, 0x89, 0x00,\
0x8a, 0x8a, 0x8a, 0x00, 0x8b, 0x8b,\
0x8b, 0x00, 0x8c, 0x8c, 0x8c, 0x00,\
0x8d, 0x8d, 0x8d, 0x00, 0x8e, 0x8e,\
0x8e, 0x00, 0x8f, 0x8f, 0x8f, 0x00,\
0x90, 0x90, 0x90, 0x00, 0x91, 0x91,\
0x91, 0x00, 0x92, 0x92, 0x92, 0x00,\
0x93, 0x93, 0x93, 0x00, 0x94, 0x94,\
0x94, 0x00, 0x95, 0x95, 0x95, 0x00,\
0x96, 0x96, 0x96, 0x00, 0x97, 0x97,\
0x97, 0x00, 0x98, 0x98, 0x98, 0x00,\
0x99, 0x99, 0x99, 0x00, 0x9a, 0x9a,\
0x9a, 0x00, 0x9b, 0x9b, 0x9b, 0x00,\
0x9c, 0x9c, 0x9c, 0x00, 0x9d, 0x9d,\
0x9d, 0x00, 0x9e, 0x9e, 0x9e, 0x00,\
0x9f, 0x9f, 0x9f, 0x00, 0xa0, 0xa0,\
0xa0, 0x00, 0xa1, 0xa1, 0xa1, 0x00,\
0xa2, 0xa2, 0xa2, 0x00, 0xa3, 0xa3,\
0xa3, 0x00, 0xa4, 0xa4, 0xa4, 0x00,\
0xa5, 0xa5, 0xa5, 0x00, 0xa6, 0xa6,\
0xa6, 0x00, 0xa7, 0xa7, 0xa7, 0x00,\
0xa8, 0xa8, 0xa8, 0x00, 0xa9, 0xa9,\
0xa9, 0x00, 0xaa, 0xaa, 0xaa, 0x00,\
0xab, 0xab, 0xab, 0x00, 0xac, 0xac,\
0xac, 0x00, 0xad, 0xad, 0xad, 0x00,\
0xae, 0xae, 0xae, 0x00, 0xaf, 0xaf,\
```

```
0xaf, 0x00, 0xb0, 0xb0, 0xb0, 0x00,\
0xb1, 0xb1, 0xb1, 0x00, 0xb2, 0xb2,\
0xb2, 0x00, 0xb3, 0xb3, 0xb3, 0x00,\
0xb4, 0xb4, 0xb4, 0x00, 0xb5, 0xb5,\
0xb5, 0x00, 0xb6, 0xb6, 0xb6, 0x00,\
0xb7, 0xb7, 0xb7, 0x00, 0xb8, 0xb8,\
0xb8, 0x00, 0xb9, 0xb9, 0xb9, 0x00,\
0xba, 0xba, 0xba, 0x00, 0xbb, 0xbb,\
0xbb, 0x00, 0xbc, 0xbc, 0xbc, 0x00,\
0xbd, 0xbd, 0xbd, 0x00, 0xbe, 0xbe,\
0xbe, 0x00, 0xbf, 0xbf, 0xbf, 0x00,\
0xc0, 0xc0, 0xc0, 0x00, 0xc1, 0xc1,\
0xc1, 0x00, 0xc2, 0xc2, 0xc2, 0x00,\
0xc3, 0xc3, 0xc3, 0x00, 0xc4, 0xc4,\
0xc4, 0x00, 0xc5, 0xc5, 0xc5, 0x00,\
0xc6, 0xc6, 0xc6, 0x00, 0xc7, 0xc7,\
0xc7, 0x00, 0xc8, 0xc8, 0xc8, 0x00,\
0xc9, 0xc9, 0xc9, 0x00, 0xca, 0xca,\
0xca, 0x00, 0xcb, 0xcb, 0xcb, 0x00,\
0xcc, 0xcc, 0xcc, 0x00, 0xcd, 0xcd,\
0xcd, 0x00, 0xce, 0xce, 0xce, 0x00,\
0xcf, 0xcf, 0xcf, 0x00, 0xd0, 0xd0,\
0xd0, 0x00, 0xd1, 0xd1, 0xd1, 0x00,\
0xd2, 0xd2, 0xd2, 0x00, 0xd3, 0xd3,\
0xd3, 0x00, 0xd4, 0xd4, 0xd4, 0x00,\
0xd5, 0xd5, 0xd5, 0x00, 0xd6, 0xd6,\
0xd6, 0x00, 0xd7, 0xd7, 0xd7, 0x00,\
0xd8, 0xd8, 0xd8, 0x00, 0xd9, 0xd9,\
0xd9, 0x00, 0xda, 0xda, 0xda, 0x00,\
0xdb, 0xdb, 0xdb, 0x00, 0xdc, 0xdc,\
0xdc, 0x00, 0xdd, 0xdd, 0xdd, 0x00,\
0xde, 0xde, 0xde, 0x00, 0xdf, 0xdf,\
0xdf, 0x00, 0xe0, 0xe0, 0xe0, 0x00,\
0xe1, 0xe1, 0xe1, 0x00, 0xe2, 0xe2,\
0xe2, 0x00, 0xe3, 0xe3, 0xe3, 0x00,\
0xe4, 0xe4, 0xe4, 0x00, 0xe5, 0xe5,\
0xe5, 0x00, 0xe6, 0xe6, 0xe6, 0x00,\
0xe7, 0xe7, 0xe7, 0x00, 0xe8, 0xe8,\
0xe8, 0x00, 0xe9, 0xe9, 0xe9, 0x00,\
0xea, 0xea, 0xea, 0x00, 0xeb, 0xeb,\
0xeb, 0x00, 0xec, 0xec, 0xec, 0x00,\
0xed, 0xed, 0xed, 0x00, 0xee, 0xee,\
0xee, 0x00, 0xef, 0xef, 0xef, 0x00,\
0xf0, 0xf0, 0xf0, 0x00, 0xf1, 0xf1,\
0xf1, 0x00, 0xf2, 0xf2, 0xf2, 0x00,\
```

```
                                0xf3, 0xf3, 0xf3, 0x00, 0xf4, 0xf4,\
                                0xf4, 0x00, 0xf5, 0xf5, 0xf5, 0x00,\
                                0xf6, 0xf6, 0xf6, 0x00, 0xf7, 0xf7,\
                                0xf7, 0x00, 0xf8, 0xf8, 0xf8, 0x00,\
                                0xf9, 0xf9, 0xf9, 0x00, 0xfa, 0xfa,\
                                0xfa, 0x00, 0xfb, 0xfb, 0xfb, 0x00,\
                                0xfc, 0xfc, 0xfc, 0x00, 0xfd, 0xfd,\
                                0xfd, 0x00, 0xfe, 0xfe, 0xfe, 0x00,\
                                0xff, 0xff, 0xff, 0x00,

    };

    int i,i2;
    char tmp;
    char filename[60]="\0";
    char *newname;
    char template[20]="frameXXXXXX";
    //BW or Color image
    int format;
    //Demensions of image
    int width;
    int height;
    int align = 0;  // needed for both types of images
    int padding =0; // needed by color images

    //We pick based on image size. We assume that images can
    //either be 3 byte RGB or 1 byte Black and White

    //Black and white images
    if((myBufSize == 1024)||(myBufSize == 4096)||
                            (myBufSize == 16384)) {
        format = 0;
    }
    //RGB color images
    else if ((myBufSize == 3072)||(myBufSize ==12288)||
                                  (myBufSize == 49152))  {
        format = 1;
    }
    //We cannot create an image, the buffer size is not correct
    else {
        return;
    }

    switch(myBufSize) {
    case 1024:
```

```
case 3072:
    width = 32;
    height = 32;
    break;
case 4096:
case 12288:
    width = 64;
    height = 64;
    break;
case 16384:
case 49152:
    width = 128;
    height = 128;
}

// Open a file to save data
//newname=mktemp(template);
//strcat(filename,newname);
//strcat(filename,".bmp");
sprintf(filename,"frame-%i.bmp",time(NULL));
if((fp = fopen (filename, "wb"))==NULL)
    {
        printf("Output Data file error\n");
        exit(1);
    }

/********Black/White image*********/
if (format == 0){
    align = ((height*width)+54) % 4;

    h1.type = 19778; //'M' * 256 + 'B';
    h1.size = (width*height) + 54 + align;
    h1.reserved1 = 0;
    h1.reserved2 = 0;
    h1.offset = 54+256*4;

    h2.size = 40;
    h2.width = width;
    h2.height = height;
    h2.planes = 1;
    h2.bits = 8;
    h2.compression = 0;
    h2.imagesize = 0;
    h2.xresolution = 0;
    h2.yresolution = 0;
```

```
        h2.ncolours = 256;
        h2.importantcolours = 0;
    }

    /************COLOR image**********/
    if (format == 1){
        h1.type = 19778; //'M' * 256 + 'B';
        h1.size = (width*height*3) + 54;
h1.reserved1 = 0;
        h1.reserved2 = 0;
        h1.offset = 54;

        h2.size = 40;
        h2.width = width;
        h2.height = height;
        h2.planes = 1;
        h2.bits = 24;
        h2.compression = 0;
        h2.imagesize = 0;
        h2.xresolution = 0;
        h2.yresolution = 0;
        h2.ncolours = 0;
        h2.importantcolours = 0;

i2=1;
//This is necessary to get the RGB values in the right
//  order for BMP.  Apparently cyclops does not provide
  //  the appropriate ordering of the RGB values for BMP
//  conversion
     for(i=0;i<width*height*3; i++)
     {
          if (i2 % 3 == 0)
         {
           tmp = myBuf[i];
           myBuf[i] = myBuf[i-2];
           myBuf[i-2]=tmp;
           }
          i2++;
     }
    }

    fwrite (&h1, sizeof(HEADER), 1, fp);
    fwrite (&h2, sizeof(INFOHEADER), 1, fp);

    if (format == 0)  //Insert Color Pallete only if BW image
```

```
        {
            fwrite (palette, sizeof(char), 256*4, fp);
            fwrite (myBuf, sizeof(char), width*height, fp);
        }

    if (format == 1)
        fwrite (myBuf, sizeof(char), width*height*3, fp);

    fclose(fp);

}

//*************************************************************
//This function prints progress
//*************************************************************
void print_progress(TOS_SERIAL_Msg myMsg)
{
    printf(".");fflush(stdout);
    return;
}


//*************************************************************
//This function returns amount of necessary data for the dump
//*************************************************************
int getSize(TOS_SERIAL_Msg myMsg)
{
    serialDumpHeader_t myHeader;
    memcpy(&myHeader,&myMsg.data,sizeof(serialDumpHeader_t));
    printf("\nreceiving buffer of size:%d ->",myHeader.seq);
    fflush(stdout);
    //printf("received buffer size:%02x:%02x\n",
    //        (char)(myHeader.seq),(char)(myHeader.seq>>8));
    return myHeader.seq;
}

//*************************************************************
//This function returns amount of necessary data for the dump
//*************************************************************
mode append(char *buf,TOS_SERIAL_Msg myMsg,int size)
{
    serialDumpHeader_t myHeader;
    int i;
    memcpy(&myHeader,&myMsg.data,sizeof(serialDumpHeader_t));
    //printf("\nheadersize %d, Msg size %d, data size %d,"
```

```
    //sizeof(serialDumpHeader_t),sizeof(myMsg),
    //sizeof(myMsg.data));
    //printf("sequence no:%d (%02x %02x)\n",myHeader.seq,
    //      (myHeader.seq&0xff00)>>8,(myHeader.seq&0x00ff) );

    if(myHeader.seq>size) return PACKET_ERROR;

    if( myHeader.seq >
               (TOSH_DATA_LENGTH-sizeof(serialDumpHeader_t)) )
               //this is not the last packet
        {
            memcpy(buf+(size-myHeader.seq),
                  myMsg.data+sizeof(serialDumpHeader_t),
                  (TOSH_DATA_LENGTH-sizeof(serialDumpHeader_t)));
            return GET_NEXT_PACKET;
        }
    else  //this is the last packet
        {
            memcpy(buf+(size-myHeader.seq),
                  myMsg.data+sizeof(serialDumpHeader_t),
                  myHeader.seq);
            return IDLE;
        }
}

//*****************************************************************
//Main
//*****************************************************************

int main(int argc, char ** argv)
{
    FILE *tsfp;
    char tsfilename[100];
    int n = 0;
    unsigned long packetcount = 0;
    unsigned long oldcount = 0;
    unsigned char *buf;
    int buffSize;
    mode myMode=IDLE;
    bool verberose= FALSE;

    if (argc > 4 || argc > 1 && argv[1][0] == '-') {
        print_usage(argv[0]);
        exit(2);
    }
```

```
if (argc==4 && argv[3][1]=='t') {
    verberose=TRUE;
} else if(argc==4) {
    print_usage(argv[0]);
    exit(2);
}
open_input(argc, argv);

sprintf(tsfilename,"timestamp-%u.dat",time(NULL));
if((tsfp = fopen (tsfilename,"wb"))==NULL)
    {
        printf("Output Data file error\n");
        exit(1);
    }

//Here we run the state machine forever
while(1) {
    TOS_SERIAL_Msg msg;
    msg = read_packet();
    if(myMode==IDLE && msg.type==IMG_MSG_TYPE)
        {
            buffSize=getSize(msg);
            if(buffSize>0)
                {
                    myMode=GET_NEXT_PACKET;
                    buf = (char *) malloc(buffSize);

                    bzero(buf,buffSize);
                }
        }
    else if(msg.type==IMG_MSG_TYPE)
    {
       if(verberose) print_packet(msg);
       else print_progress(msg);
       //if(verberose) print_data(msg);
       switch(append(buf,msg,buffSize))
           {
           case GET_NEXT_PACKET:
               packetcount++;
               if(packetcount<oldcount+100000){
                    fprintf(tsfp,"%u S:%i O:%i %u %u %i\n",
                            time(NULL),msg.sourceaddr,
                            msg.originaddr,packetcount,
                            msg.seqno,(int8_t)msg.rssi-45);
             fflush(tsfp);
```

```
   } else {
fclose(tsfp);
sprintf(tsfilename,"timestamp-%u.dat",
                                time(NULL));
     if((tsfp = fopen (tsfilename,"wb"))==NULL
)
         {
                              printf("Output Data file error\n");
                              exit(1);
         }
                        fprintf(tsfp,"%u S:%i O:%i %u %u %i\n",
                                time(NULL),msg.sourceaddr,
                                msg.originaddr,packetcount,
                                msg.seqno,(int8_t)msg.rssi-45);
                  fflush(tsfp);
                        oldcount=packetcount;
   }
                    break;
                case IDLE:
                    packetcount++;
                    if(packetcount<oldcount+100000){
                        fprintf(tsfp,"%u S:%i O:%i %u %u %i\n",
                                time(NULL),msg.sourceaddr,
                                msg.originaddr,packetcount,
                                msg.seqno,(int8_t)msg.rssi-45);
                  fflush(tsfp);
   } else {
    fclose(tsfp);
sprintf(tsfilename,"timestamp-%u.dat",
                                time(NULL));
     if((tsfp = fopen(tsfilename,"wb"))==NULL)
         {
                              printf("Output Data file error\n");
                              exit(1);
         }
                        fprintf(tsfp,"%u S:%i O:%i %u %u %i\n",
                                time(NULL),msg.sourceaddr,
                                msg.originaddr,packetcount,
                                msg.seqno,(int8_t)msg.rssi-45);
                  fflush(tsfp);
                        oldcount=packetcount;
   }
                    print_dump(buf,buffSize);
                    print_file_dump(buf,buffSize);
                    print_bmp_dump(buf,buffSize);
```

```
                    printf("received packets: %i\n",packetcount);
                    printf("sent packets: %i\n",msg.seqno);
                    myMode=IDLE;
                    free(buf);
                    break;
                case PACKET_ERROR:
                default:
                    packetcount++;
                    if(packetcount<oldcount+100000){
                         fprintf(tsfp,"%u S:%i O:%i %u %u %i\n",
                                 time(NULL),msg.sourceaddr,
                                 msg.originaddr,packetcount,
                                 msg.seqno,(int8_t)msg.rssi-45);
                  fflush(tsfp);
                    }
   else {
    fclose(tsfp);
sprintf(tsfilename,"timestamp-%u.dat",
                                 time(NULL));
     if((tsfp = fopen(tsfilename,"wb"))==NULL)
         {
                    printf("Output Data file error\n");
                    exit(1);
         }
                         fprintf(tsfp,"%u S:%i O:%i %u %u %i\n",
                                 time(NULL),msg.sourceaddr,
                                 msg.originaddr,packetcount,
                                 msg.seqno,(int8_t)msg.rssi-45);
                  fflush(tsfp);
                         oldcount=packetcount;
   }
                    printf("!");
                    //printf("an error encountered\n");
                    //myMode=IDLE;
                }
       }
       else {
           // this is not an image message
           if(verberose) print_packet(msg); else printf("r");
           fflush(stdout);
       }
   }
}
```

I.2 def.h

```
//Modified by Paul Bender bender.13@wright.edu to work with MICAz
//                        using TinyOS multihop routing.

#ifndef DEF_H
#define DEF_H

#define BAUDRATE B4800 //the default baudrate that the
               //device is talking
#define SERIAL_DEVICE "/dev/ttyS0" //the port to use.

typedef unsigned char bool;
#define TRUE 1
#define FALSE 0

typedef enum {IDLE=12,GET_NEXT_PACKET,PACKET_ERROR} mode;
#define TOSH_DATA_LENGTH 22
#define uint8_t unsigned char
#define uint16_t unsigned short
#define IMG_MSG_TYPE 0x17
//#include "AM.h"

//I hate to do that but Unfortunately there is no data
// structure I can grab from AM.h that has the serial
// port content. I am sorry so I had to create a data
// structure here. This means that if the actual data
// structure change this code will be broken. But this
// is very unlikely.
typedef struct TOS_SERIAL_Msg
{
  /* TOS_MSG data*/
  uint16_t addr;
  uint8_t type;
  uint8_t group;
  //uint8_t length;
  /*Payload */
  uint16_t sourceaddr;
  uint16_t originaddr;
  int16_t seqno;
  int8_t hopcount;
  int8_t data[TOSH_DATA_LENGTH];
  int8_t rssi; // RSSI value +45, subtract 45 to get the
               // real value - PAB
```

```
  uint16_t crc;
}TOS_SERIAL_Msg;

int input_stream;

void print_usage(char *name);
void open_input(int argc, char **argv);
TOS_SERIAL_Msg read_packet();
int getSize(TOS_SERIAL_Msg myMsg);
void print_progress(TOS_SERIAL_Msg myMsg);

#endif
```

Appendix J:Multi-Hop Mote with Integrated Serial Serial Base

J.1 MoteI2CRelayC.nc

```
includes MoteI2CRelay;
includes CyclopsNetwork;
includes MultiHop;

// Modified by Paul Bender (bender.13@wright.edu) to use MICAz
// motesto send Cyclops images over multiple hops to the sink
// node using the reliableroute module.  This version also
// integrates into the code written to all motes.

configuration MoteI2CRelayC { }
implementation {
    components
        Main,
        MoteI2CRelayM,
        CC2420RadioC,
        TimerC, I2CPacketMasterC,
        Bcast,
        GenericCommPromiscuous as Comm,
        //WMEWMAMultiHopRouter as multihopM,
        EWMAMultiHopRouter as multihopM,
        //MultiHopRouter as multihopM,
        QueuedSend,
        LedsC,
        UARTNoCRCPacket;

    Main.StdControl -> MoteI2CRelayM;
    Main.StdControl -> TimerC;
    Main.StdControl -> Bcast.StdControl;
    Main.StdControl -> multihopM.StdControl;
    Main.StdControl -> QueuedSend.StdControl;
```

```
    Main.StdControl -> Comm;

    MoteI2CRelayM.UARTControl -> UARTNoCRCPacket;
    MoteI2CRelayM.UARTSend -> UARTNoCRCPacket;
    MoteI2CRelayM.UARTReceive -> UARTNoCRCPacket;

    MoteI2CRelayM.Leds -> LedsC;
    MoteI2CRelayM.Timer -> TimerC.Timer[unique("Timer")];

    //RF communication facility
    //MoteI2CRelayM.CommControl -> Comm;
    MoteI2CRelayM.PowerControl -> CC2420RadioC;
    MoteI2CRelayM.MacControl-> CC2420RadioC;
    MoteI2CRelayM.SendMsg -> Comm.SendMsg[AM_CYCLOPSMHOPCMDMSG];
    //MoteI2CRelayM.ReceiveMsg -> Comm.ReceiveMsg[Sample_Packet];


    MoteI2CRelayM.Bcast -> Bcast.Receive[AM_CYCLOPSMHOPCMDMSG];
    Bcast.ReceiveMsg[AM_CYCLOPSMHOPCMDMSG] ->
                          Comm.ReceiveMsg[AM_CYCLOPSMHOPCMDMSG];

    multihopM.ReceiveMsg[AM_CYCLOPSMHOPCMDMSG] ->
                          Comm.ReceiveMsg[AM_CYCLOPSMHOPCMDMSG];

    MoteI2CRelayM.Intercept->
                     multihopM.Intercept[AM_CYCLOPSMHOPDATAMSG];

    MoteI2CRelayM.RouteControl -> multihopM;
    MoteI2CRelayM.Send -> multihopM.Send[AM_CYCLOPSMHOPDATAMSG];
    multihopM.ReceiveMsg[AM_CYCLOPSMHOPDATAMSG] ->
                         Comm.ReceiveMsg[AM_CYCLOPSMHOPDATAMSG];

    //Communcation over I2C
    MoteI2CRelayM.I2CPacketMaster ->
           I2CPacketMasterC.I2CPacketMaster[CYCLOPS_I2C_ADDRESS];
    MoteI2CRelayM.I2CStdControl -> I2CPacketMasterC.StdControl;

}
```

J.2 MoteI2CRelayM.nc

```
// Modified by Paul Bender (bender.13@wright.edu) to use MICAz
```

```
// motesto send Cyclops images over multiple hops to the sink
// node using the reliableroute module.  This version also
// integrates into the code written to all motes.

module MoteI2CRelayM {
    provides interface StdControl;
    //provides command result_t getNextPacket(uint8_t offset);
    uses {
        interface Leds;

        //Serial Communications
        interface StdControl as UARTControl;
        interface BareSendMsg as UARTSend;
        interface ReceiveMsg as UARTReceive;


        //RF communication
        interface StdControl as CommControl;
        interface SendMsg as SendMsg;
        //interface ReceiveMsg as ReceiveMsg;
        interface Send as Send;
        interface CC2420Control as PowerControl;
        interface Receive as Bcast;
        interface RouteControl;
        interface Intercept;
        interface MacControl;

        //I2C communication
        interface I2CPacketMaster;
        interface StdControl as I2CStdControl;

        //Timer
        interface Timer;
    }
}
implementation {
    TOS_Msg msg1;             /* Message to be sent out */
    TOS_Msg msg2;             /* ack Message to be sent out */
    uint16_t length;               /* length of the payload */
    CyclopsPacket request;    /* Request to be sent to Cyclops*/
    bool CameraOn = TRUE;
    bool TimerRunning = FALSE;
    bool UARTBusy = FALSE;
    uint16_t sequence;
```

```
  enum {
    SERIAL_QUEUE_SIZE = 16, // Serial Port Queue
    EMPTY = 0xff
  };

  /* Internal storage for serial port queue and scheduling state
*/
  TOS_Msg SerialBuffers[SERIAL_QUEUE_SIZE];
  TOS_MsgPtr SerialBufList[SERIAL_QUEUE_SIZE];

  uint8_t iSerialBufHead, iSerialBufTail;
  bool BufferFull = FALSE;

  static void initialize() {
    int n;

    for (n=0; n < SERIAL_QUEUE_SIZE; n++) {
      SerialBufList[n] = &SerialBuffers[n];
    }

    iSerialBufHead = iSerialBufTail = 0;
  }


    command result_t StdControl.init() {
       //Initialize Radio stack
       //call CommControl.init();
       //Initialize host-side of I2C
       call I2CStdControl.init();
       //Initialize leds
       call Leds.init();
       //Initialize UART
       initialize();
       call UARTControl.init();
       return SUCCESS;
    }
    command result_t StdControl.start() {
        //Start radio stack
        //call CommControl.start();
       /* Set the RF Power. Argument between 3 (lowest) and
          31 (highest, default)*/
        call PowerControl.SetRFPower(31);
        /* set the RF channel to the desired channel.
           NOTE: Must match base
           NOTE: Chanel 25 and 26 do not interfere with 802.11
```

```
         */
        //call PowerControl.TunePreset(26);
        /* turn automatic acknowledgements on */
        //call PowerControl.enableAutoAck();
        //call MacControl.enableAck();

        // set the update frequency
        call RouteControl.setUpdateInterval(30);

        //Start host-side of I2C
        call I2CStdControl.start();
           /*************************************************
             *PARAMETERS TO SET:
             * request.framesize
             *    Sets the output image size
             *    Possible values:
             *         32  <= 32x32 image
             *         64  <= 64x64 image
             *         128 <= 128x128 image
             * request.type
             *    Sets the type of output image (color,
             *                                   black&white,ect)
             *    Possible values:
             *         CYCLOPS_IMAGE_TYPE_Y     <= Black and
             *                                     white image. 1
             *                                     byte per pixel.
             *         CYCLOPS_IMAGE_TYPE_RGB   <= Color image. 3
             *                                     bytes per pixel.
             *         CYCLOPS_IMAGE_TYPE_YCbCr <= Color image. 2
             *                                     bytes per pixel.
             *
             * NOTE: EXTENSIVE TESTING WAS NOT DONE WITH
             *       CYCLOPS_IMAGE_TYPE_YCbCr
             *************************************************/

        request.frameSize = 128;
        request.type = CYCLOPS_IMAGE_TYPE_Y;

        //Start timer
        TimerRunning=FALSE;
        call Timer.start(TIMER_ONE_SHOT, 10000);

        return SUCCESS;
    }
```

```
    command result_t StdControl.stop() {
        //Stop the radio stack
        //call CommControl.stop();
        //Stop host-side of I2C
        call I2CStdControl.stop();
        return SUCCESS;
    }

    event result_t Timer.fired() {
        uint16_t parent;
        call Leds.greenOn();
        if((parent=call RouteControl.getParent())==0xffff)
{
           // manually trigger a routing update
           call RouteControl.manualUpdate();
           call Leds.greenOff();
        }
        else {
        //Write packet to cyclops over I2C
        sequence=128*128;
        //if((TimerRunning==FALSE))
//{
        //   TimerRunning=TRUE;
        //   call Timer.start(TIMER_REPEAT,60000);
                                        // every 60 seconds - PAB
//}
        call I2CPacketMaster.writePacket(sizeof(request),
                                          (char*)&request);
        }
        return SUCCESS;
    }

    /*************************Communication Stack*************/
    task void sendData()
    {
       call Send.send(&msg1,length);
    }
    //command result_t getNextPacket(uint8_t offset) {
    task void getNextPacket() {
        uint8_t *pData;
        call Leds.yellowToggle();

        //If at least one more full packet remains for the image
        if(sequence >= (2*CYCLOPS_DATA)) {
            if ((pData =
```

```
                (uint8_t *)call Send.getBuffer(&msg1,&length)))
            {
               call I2CPacketMaster.readPacket(length,pData);
            }
        }
        //If less than a full packet remains for the image
        else if (sequence > (CYCLOPS_DATA)) {
           if ((pData =
               (uint8_t *)call Send.getBuffer(&msg1,&length)))
           {
               call I2CPacketMaster.readPacket(
                                    (sequence-CYCLOPS_DATA)+
                                    sizeof(serialDumpHeader_t),
                                    pData);
           }
        }
        else
           call Timer.start(TIMER_ONE_SHOT, 60000);
        sequence=sequence-CYCLOPS_DATA;
    }


    event result_t Send.sendDone(TOS_MsgPtr sent,
                                 result_t success) {
        uint16_t randtime;
        //serialDumpHeader_t* dataPtr =
        //          (serialDumpHeader_t*)sent->data;
        randtime=rand()*5000;
TOSH_uwait(randtime+25000);
        //if(sent->ack!=1)
//post sendData(); //resend the data if not acknowledged
        //else {
          // grab the next packet
  post getNextPacket();
        //}
        return SUCCESS;
    }

    event result_t SendMsg.sendDone(TOS_MsgPtr sent,
                                    result_t success) {
        return SUCCESS;
    }
    event TOS_MsgPtr Bcast.receive(TOS_MsgPtr pMsg,
                                   void* payload,
                                   uint16_t payloadLen){
```

```
 CyclopsCmdMsg *pCmdMsg = (CyclopsCmdMsg *)payload;

 if(pCmdMsg->type == CYCLOPS_TYPE_ACK) {
    if(pCmdMsg->parentaddr==TOS_LOCAL_ADDRESS &&
          pCmdMsg->args.sequenceNo == sequence ) {
post getNextPacket();
     }
 }

 if (pCmdMsg->type == CYCLOPS_TYPE_CAMERAOFF ) {
   CameraOn = FALSE ;
   // Need to implement, send turn power off to camera
   call Timer.stop();

 } else if (pCmdMsg->type == CYCLOPS_TYPE_CAMERAON) {

     // Need to impelment, send turn power on to camera
     CameraOn = TRUE;
 } else if (pCmdMsg->type == CYCLOPS_TYPE_SETSIZE ) {
   // Set the size of the image
   request.frameSize=pCmdMsg->args.imagesize;
 } else if (pCmdMsg->type == CYCLOPS_TYPE_SETTYPE ) {
   // Set the type of the image
   request.type = pCmdMsg->args.imagetype;
 }
     return pMsg;
 }

 //event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr pMsg) {
 //    return pMsg;
 //}
 /*****************************I2C*****************************/
 event result_t I2CPacketMaster.readPacketDone(char len,
                                               char *data) {
     //if(len>0){
        //sequence=sequence-CYCLOPS_DATA;
post sendData();
        return SUCCESS;
     //} else {
     //   post getNextPacket();
     //   return FAIL;
     //}
 }
 event result_t I2CPacketMaster.writePacketDone(bool r) {
```

```
        uint8_t *pData;
        if(r == FAIL) {
            call Leds.redToggle();
            return FAIL;
        }
        //Read packet from I2C
        if ((pData =
             (uint8_t *)call Send.getBuffer(&msg1,&length)))
        {
           call I2CPacketMaster.readPacket(length, pData);
           return SUCCESS;
        }
        else return FAIL;
   }

void insert_buffer(TOS_MsgPtr ptr){
    int8_t n;
    call Leds.yellowOn();
    if(iSerialBufHead == iSerialBufTail && BufferFull)
    {
return;  // Drop this one.
    }
    else {
        atomic{
           SerialBufList[iSerialBufHead]=ptr;
           iSerialBufHead=(iSerialBufHead+1 % SERIAL_QUEUE_SIZE);
           if(iSerialBufHead==iSerialBufTail){
                BufferFull=TRUE;
                call Leds.redOn();
           }
        }
     }
  }

  int8_t get_buffer(){
    int8_t n;
    atomic{
      n = iSerialBufTail;
      iSerialBufTail=(iSerialBufTail+1 % SERIAL_QUEUE_SIZE);
      BufferFull=FALSE;
      call Leds.redOff();
    }
  }

task void SendSerial(){
```

```
     if(!UARTBusy) {
        if(iSerialBufHead!=iSerialBufTail || BufferFull) {
           int myIndex=get_buffer();
   TOS_MsgPtr data=SerialBufList[myIndex];
           UARTBusy=TRUE;
           data->addr = TOS_UART_ADDR;
           call UARTSend.send(data);
        }
      } else post SendSerial();
}


   // Intercept data from radio
event result_t Intercept.intercept(TOS_MsgPtr data,void* payload,
                                              uint16_t payloadLen){
       //if(data->addr==TOS_LOCAL_ADDRESS) {
        //acknowledge the message
        //uint16_t mylength;
        //CyclopsCmdMsg *pCmdMsg =
        // (CyclopsCmdMsg *)call Send.getBuffer(&msg2,&mylength);
        //TOS_MHopMsg *pDataMsg = (TOS_MHopMsg *)payload;
        //pCmdMsg->type = CYCLOPS_TYPE_ACK;
        //pCmdMsg->parentaddr= pDataMsg->sourceaddr;
        //pCmdMsg->args.sequenceNo = pDataMsg->seqno;
        //call SendMsg.send(TOS_BCAST_ADDR,pCmdMsg,mylength);
       //}
       if(TOS_LOCAL_ADDRESS==0)
       {
          /*if(data->addr==TOS_LOCAL_ADDRESS && !UARTBusy)
          {
            UARTBusy=TRUE;
            call Leds.greenToggle();
            data->addr = TOS_UART_ADDR;
            call UARTSend.send(data);
            return FAIL ;
          }
          else */
          if(data->addr==TOS_LOCAL_ADDRESS)
          {
    // Buffer this data
            insert_buffer(data);
            post SendSerial();
            return FAIL;
          }
          return FAIL;
```

```
      }
         else
           return SUCCESS;

  }

  event result_t UARTSend.sendDone(TOS_MsgPtr msg,
                                   result_t success) {
    UARTBusy=FALSE;
    return SUCCESS;
  }

  event TOS_MsgPtr UARTReceive.receive(TOS_MsgPtr data) {
    return data;
  }

}
```